MULTISERVICE NETWORK PROCESSING DEVICE

Bidyut Parruck
Chulanur Ramakrishnan
Rami Zecharia
Nael Atallah
Hung Nguyen
Shankar Channabasappa

CROSS-REFERENCE TO RELATED APPLICATIONS

This application claims priority under 35 U.S.C. 120 from the following U.S. patent applications: 1) U.S. Patent Application Serial No. 09/528,802, filed March 20, 2000, 2) U.S. Patent Application Serial No. 09/539,479, filed March 30, 2000, 3) U.S. Patent Application Serial No. 09/539,306, filed March 30, 2000, 4) U.S. Patent Application Serial No. 09/539,478, filed March 30, 2000, 5) U.S. Patent Application Serial No. 09/539,461, filed March 30, 2000, 6) U.S. Patent Application Serial No. 09/539/476, filed March 30, 2000, and 7) U.S. Patent Application Serial No. 09/539,477, filed March 30, 2000. The subject matter of the above-listed seven patent applications is incorporated herein by reference.

BACKGROUND INFORMATION

The Maximus and Internet Growth

As the Internet grows, the volume of data traversing the various networks that comprise the Internet has increased tremendously. While packet-based technology is widely believed to be the dominant technology going forward, in the short term, no single technology has emerged as the clear winner in the race to satisfy the seemingly insatiable demand for bandwidth, which is fueled by the spectacular growth of the Internet.

At the same time, carriers respond to customers' demand for new applications by rolling out increasingly sophisticated data services, some of which are highly delay-sensitive and/or bandwidth-intensive (e.g., voice or video). The introduction of these differentiated services opens up opportunities for competing system vendors to introduce new technologies and/or protocols. As a consequence, changes in the networks are characterized not only by vastly increased bandwidth requirements but also by a highly diverse array of technologies and protocols.

The aforementioned changes impact the **access**, **core/long-haul**, and **metro/regional** networks in different ways. In the foreseeable future, the demand for high speed **access** (i.e., in the megabit to low gigabit range) appears to be satisfied by a multitude of competing technologies, e.g., xDSL, cable modem, Frame Relay, T1, T3, ATM, and Gigabit Ethernet. Some of the newer technologies such as xDSL and cable modem have made impressive gains, enabling users to transmit and receive data at vastly increased data rates, thereby facilitating higher-bandwidth applications, some which were impractical only a few years ago when dial-up or lower speed ISDN connections were the dominant access technologies.

Corresponding changes also occur in the **core/long-haul** networks. In the core network, the emergence of an all-optical core and DWDM technologies have made it practical to transfer terabits of data across long-haul trunks at substantially lower costs.

However, all-optical technologies have not progressed to the point where it is possible to process packets and/or cells in the optical domain. Processing intelligence is still very much the province of electronics. Accordingly, an all-optical core, when operated in its pure optical mode, is substantially devoid of processing intelligence, such as the type required to correctly perform end-to-end routing of a packet or cell from one user to another. Without resorting to the electrical plane, it is currently not practical to offer differentiated data services (e.g., routing different streams of traffic in different ways in accordance to their classes of service) via all-optical technologies.

The **metro/regional** network bears the full brunt of the powerful changing forces that affect the access and core networks. As an aggregation mechanism for access networks, the metro/regional network must move to higher speeds to cope with the increased bandwidth requirements. The metro/regional network must also interoperate with legacy and emerging transmission technologies and protocols, as well as adapt the divergent technologies and protocols to its own transmission requirements and the requirements of the core network. The metro/regional network must also perform extensive cell and packet processing, as these tasks are no longer performed by the optical core network. To facilitate differentiated service offerings, the metro/regional network must also handle traffic management for different data streams having different classes of service. In short, the metro/regional network of tomorrow is characterized by higher speeds, diverse transmission technologies and protocols, and enhanced processing requirements.

Capabilities of the Maximus

The Maximus is designed to capitalize on these trends. It is a high integration ASIC designed for high speed switching and routing systems. The Maximus also finds applications in access devices operating at OC-192, or as an aggregation device in the core network. In a typical line card deployment, the Maximus is implemented between an OC-192 or a quad OC-48 multi-service framer and the switch fabric.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention is illustrated by way of example, and not by way of limitation, in the Figures of the accompanying drawings and in which like reference numerals refer to similar elements and in which:

Figure 1 is a figure showing multi-service system.
Figure 2 is a figure showing OC-192 line card for traffic manager application.
Figure 3 is a figure showing OC-192 line card for traffic manager application.
Figure 4 is a figure showing various applications of the maximus chipset along with the type field.
Figure 5 is a figure showing traffic manager applications.
Figure 5A is a figure showing uses of the Maximus chip with cell-based switch fabric.
Figure 6 is a figure showing SAR applications.
Figure 6A is a figure showing uses of the Maximus chip with packet-based switch fabric.
Figure 7 is a figure showing ingress tm (ATM=>ATM).
Figure 8 is a figure showing ingress tm (ATM=> MPLS packet).
Figure 9 is a figure showing ingress tm (MPLS packet=>ATM).

Figure 10 is a figure showing ingress tm (packet=>packet).
Figure 11 is a figure showing ATM encapsulation.
Figure 12 is a figure showing reassembly.
Figure 13 is a figure showing ingress packet bypass.
Figure 14 is a figure showing egress tm (ATM=>ATM).
Figure 15 is a figure showing egress tm (ATM=> MPLS packet).
Figure 16 is a figure showing egress tm (MPLS packet =>ATM).
Figure 17 is a figure showing egress tm (packet=>packet).
Figure 18 is a figure showing ATM de-encapsulation.
Figure 19 is a figure showing segmentation.
Figure 20 is a figure showing egress packet bypass.
Figure 21 is a figure showing Maximus chipset top level block diagram.
Figure 22 is a figure showing signals.
Figure 23 is a figure showing block addresses.
Figure 24 is a figure showing read access and timings definitions.
Figure 25 is a figure showing read access characteristic parameters.
Figure 26 is a figure showing write access and timings definitions.
Figure 27 is a figure showing write access characterisitcs parameters.
Figure 28 is a figure showing interface from the CPU if to the blocks.
Figure 29 is a figure showing global sync.
Figure 30 is a figure showing slot_count and CPU_port signals.
Figure 31 is a figure showing interface of the CPU if block.
Figure 32 is a figure showing internal implementation for locking the current CPU access. parameters.
Figure 33 is a figure showing test mux block diagram.
Figure 34 is a figure showing soft resets.
Figure 35 is a figure showing input/output disable.
Figure 36 is a figure showing registers.
Figure 37 is a figure showing incoming spi-4 interface.
Figure 38 is a figure showing input control block in spii.
Figure 39 is a figure showing output control block in spii.
Figure 40 is a figure showing bit align block.
Figure 41 is a figure showing drop idle state machine.
Figure 42 is a figure showing control word format.
Figure 43 is a figure showing descriptions of fields in the control words.
Figure 44 is a figure showing control word type list.
Figure 45 is a figure showing rx state machine transition table.
Figure 46 is a figure showing extent of dip-4 codewords.
Figure 47 is a figure showing example of dip-4 encoding (odd parity).
Figure 48 is a figure showing example of a port calendar.
Figure 49 is a figure showing fifo status state machine.
Figure 50 is a figure showing example of dip-2 encoding (odd parity).
Figure 51 is a figure showing summary of internal memories.
Figure 52 is a figure showing interface ports.
Figure 53 is a figure showing register description.
Figure 54 is a figure showing outgoing spi-4 interface.

Figure 55 is a figure showing training sequence.
Figure 56 is a figure showing tx state machine transition table.
Figure 57 is a figure showing extent of dip-4 codewords.
Figure 58 is a figure showing example of dip-4 encoding (odd parity).
Figure 59 is a figure showing example of transfer data byte ordering.
Figure 60 is a figure showing tx state machine block.
Figure 61 is a figure showing port calendar.
Figure 62 is a figure showing status interface to database.
Figure 63 skew detection.
Figure 64 is a figure showing seg_table – 64 x 26 bits internal dual-port SRAM.
Figure 65 is a figure showing interface.
Figure 66 is a figure showing timing diagram-reassembly port empty.
Figure 67 is a figure showing timing diagram-back-to-back port transfer.
Figure 68 is a figure showing register description.
Figure 69 is a figure showing software interface registers.
Figure 70 is a figure showing read outgoing spi start-up parameters.
Figure 71 is a figure showing write outgoing spi start-up parameters.
Figure 72 is a figure showing read outgoing spi start-up parameters.
Figure 73 is a figure showing write outgoing spi start-up parameters.
Figure 74 is a figure showing read outgoing spi start-up parameters.
Figure 75 is a figure showing write outgoing spi start-up parameters.
Figure 76 is a figure showing write port parameters after initialization.
Figure 77 is a figure showing read port parameters after initialization.
Figure 78 is a figure showing read port full parameters.
Figure 79 is a figure showing write port full parameters.
Figure 80 is a figure showing read control status register.
Figure 81 is a figure showing write spio_tstmux_sel.
Figure 82 is a figure showing read spio_tstmux_sel.
Figure 83 is a figure showing lookup engine and neighboring blocks.
Figure 84 is a figure showing top level block diagram of the lookup engine.
Figure 85 is a figure showing data dependency on sop cell followed none sop cell on the same port.
Figure 86 is a figure showing header selection.
Figure 87 is a figure showing label hashing & record retrieval.
Figure 88 is a figure showing port parameters, it is 64 entries.
Figure 89 is a figure showing encap table structure, it is 8 entries.
Figure 90 is a figure showing port type.
Figure 91 is a figure showing record memory, it is 2meg entries per memory.
Figure 92 is a figure showing parameter width.
Figure 93 is a figure showing external interface signals.
Figure 94 is a figure showing external interface signal names and direction.
Figure 95 is a figure showing global signals.
Figure 96 is a figure showing CPU interface to look up engine.
Figure 97 is a figure showing memory size.
Figure 98 is a figure showing memory size bits.
Figure 99 is a figure showing memory chip select.

Figure 100 is a figure showing memory chip select values.
Figure 101 is a figure showing address space.
Figure 102 is a figure showing registers addresses.
Figure 103 is a figure showing nop command.
Figure 104 is a figure showing read lut memory1 command.
Figure 105 is a figure showing write lut memory1 command.
Figure 106 is a figure showing read lut memory2 command.
Figure 107 is a figure showing write lut memory2 command.
Figure 108 is a figure showing read port memory command.
Figure 109 is a figure showing write port memory command.
Figure 110 is a figure showing setup connection command for key command.
Figure 111 is a figure showing teardown connection command for key command.
Figure 112 is a figure showing get hash FID command.
Figure 113 is a figure showing get hash FID response.
Figure 114 is a figure showing read cam memory1 command.
Figure 115 is a figure showing write cam memory1.
Figure 116 is a figure showing read cam memory2.
Figure 117 is a figure showing write cam memory2.
Figure 118 is a figure showing init all memories.
Figure 119 is a figure showing init memory 1 & cam 1.
Figure 120 is a figure showing init memory 2 & cam 2.
Figure 121 is a figure showing ATM types.
Figure 122 is a figure showing ATM MPLS tag location & format.
Figure 123 is a figure showing MPLS format.
Figure 124 is a figure showing lookup types.
Figure 125 is a figure showing header format.
Figure 126 is a figure showing data position within the header format.
Figure 127 is a figure showing header substitution.
Figure 128 is a figure showing special header structure.
Figure 129 is a figure showing encap table structure.
Figure 130 is a figure showing the encapsulation header replacement in l2 case.
Figure 131 is a figure showing hashing and memory accesses.
Figure 132 is a figure showing hash timing and lookup operations.
Figure 133 is a figure showing types of segmentation and applications for various traffic types.
Figure 134 is a figure showing segmentation engine and neighboring blocks.
Figure 135 is a figure showing spi-4 data flow.
Figure 136 is a figure showing type 1 segmentation.
Figure 137 is a figure showing type 2 segmentation.
Figure 138 is a figure showing type 3 segmentation.
Figure 139 is a figure showing type 4 segmentation.
Figure 140 is a figure showing status cell access flow.
Figure 141 is a figure showing CPU access flow.
Figure 142 is a figure showing top-level block diagram for segmentation engine.
Figure 143 is a figure showing error conditions.
Figure 144 is a figure showing summary of internal memories.

Figure 145 is a figure showing seg_table – 65 x 39 bits internal dual-port SRAM
Figure 146 is a figure showing free_buffer – 80 x 7 bits internal SRAM.
Figure 147 is a figure showing q_fifo – 80 x 70 bits internal SRAM.
Figure 148 is a figure showing data SRAM – 640 x 64 bits internal SRAM.
Figure 149 is a figure showing crc_table – 65 x 32 bits internal dual-port SRAM.
Figure 150 is a figure showing statistic SRAM – 65 x 80 bits internal SRAM.
Figure 151 is a figure showing system.
Figure 152 is a figure showing segmentation ⇔ lookup.
Figure 153 is a figure showing segmentation ⇔ memory manager.
Figure 154 is a figure showing segmentation ⇔ PFQ.
Figure 155 is a figure showing segmentation ⇔ sch.
Figure 156 is a figure showing segmentation ⇔ CPU interface.
Figure 157 is a figure showing timing diagrams.
Figure 158 is a figure showing memory access.
Figure 159 is a figure showing registers.
Figure 160 is a figure showing summary of commands.
Figure 161 is a figure showing CPU inject packet.
Figure 162 is a figure showing read seg fbl memory.
Figure 163 is a figure showing write seg fbl memory.
Figure 164 is a figure showing read seg crc memory.
Figure 165 is a figure showing write seg crc memory.
Figure 166 is a figure showing read seg statistic memory.
Figure 167 is a figure showing write seg statistic memory.
Figure 168 is a figure showing read seg queue memory.
Figure 169 is a figure showing write seg queue memory.
Figure 170 is a figure showing read seg data memory.
Figure 171 is a figure showing write seg data memory.
Figure 172 is a figure showing read seg tbl memory.
Figure 173 is a figure showing write seg tbl memory.
Figure 174 is a figure showing memory manager block diagram.
Figure 175 is a figure showing internal memory manager block diagram.
Figure 176 is a figure showing multicast and BID/imBID release.
Figure 177 is a figure showing enqueue process in the internal memory manager.
Figure 178 is a figure showing dequeue process in the internal memory manager.
Figure 179 is a figure showing SDRAM manager block diagram.
Figure 180 is a figure showing memory controller configuration.
Figure 181 is a figure showing summary of memories.
Figure 182 is a figure showing summary of CAM.
Figure 183 is a figure showing segmentation engine.
Figure 184 is a figure showing per flow queue engine.
Figure 185 is a figure showing DDR-SDSRAM.
Figure 186 is a figure showing reassembly.
Figure 187 is a figure showing CPU interface.
Figure 188 is a figure showing global sync.
Figure 189 is a figure showing memory size is a figure showing 64 mbytes.
Figure 190 is a figure showing memory size is a figure showing 256 mbytes.

Figure 191 is a figure showing register description.
Figure 192 is a figure showing CPU_wr_SDRAM.
Figure 193 is a figure showing CPU_rd_SDRAM.
Figure 194 is a figure showing CPU_wr_ fiml.
Figure 195 is a figure showing CPU_rd_fiml.
Figure 196 is a figure showing CPU_wr_cam.
Figure 197 is a figure showing CPU_rd_cam.
Figure 198 is a figure showing CPU_init_mm.
Figure 199 is a figure showing inputs from segmentation.
Figure 200 is a figure showing interface with SDRAM bw optimizer is a figure showing SWM requests to write to SDRAM.
Figure 201 is a figure showing interface to SDRAM bw optimizer is a figure showing write to SDRAM.
Figure 202 is a figure showing interface with per flow queue is a figure showing receive de-queue from PFQ.
Figure 203 is a figure showing interface with SDRAM bw optimizer is a figure showing SRM requests to read from SDRAM.
Figure 204 is a figure showing interface with SDRAM bw optimizer is a figure showing SRM read from SDRAM.
Figure 205 is a figure showing interface with reassembly.
Figure 206 is a figure showing applications for various traffic types.
Figure 207 is a figure showing reassembly engine and neighboring blocks.
Figure 208 is a figure showing type 1 reassembly (ingress memory cells => switch cells).
Figure 209 is a figure showing type 1 reassembly (aal5 memory cells => switch cells).
Figure 210 is a figure showing type 1 reassembly (aal5 like memory cells => switch cells).
Figure 211 is a figure showing type 2 reassembly (52 byte ATM cells => packet).
Figure 212 is a figure showing type 3 reassembly (48 byte ATM cells => packet).
Figure 213 is a figure showing type 4 reassembly (switch cells => packet).
Figure 214 is a figure showing type 5 reassembly (52 byte ATM cells => ATM cells with translation).
Figure 215 is a figure showing type 6 reassembly (48 byte ATM cells => packet).
Figure 216 is a figure showing type 7 reassembly (48 byte cells => AAL-5 cells).
Figure 217 is a figure showing type 8 reassembly (switch cells => packet).
Figure 218 is a figure showing top level block diagram for the reassembly engine.
Figure 219 is a figure showing free buffer list block diagram.
Figure 220 is a figure showing crc_gen block diagram.
Figure 221 is a figure showing ATM MPLS tag location & format.
Figure 222 is a figure showing MPLS format.
Figure 223 is a figure showing ATM types.
Figure 224 is a figure showing seg_table – 64 x 18 bits internal single-port SRAM.
Figure 225 is a figure showing output_control_memory – 128 x 18 bit internal SRAM.
Figure 226 is a figure showing output_port_queue – 64 x 18 bit internal SRAM.
Figure 227 is a figure showing free_buffer – 128 x 7 bit internal SRAM.
Figure 228 is a figure showing data_SRAM – 1024 x 64 bit internal dual port SRAM.
Figure 229 is a figure showing header_SRAM – 256 x 64 bit internal dual port SRAM.

Figure 230 is a figure showing CPU_data_SRAM – 36 x 64 bit internal dual port SRAM.
Figure 231 is a figure showing crc_table – 64 x 32 bits internal dual-port SRAM.
Figure 232 is a figure showing statistic SRAM – 64 x 112 bit internal SRAM.
Figure 233 is a figure showing look-up SRAM – 4m x 36 bit external SRAM.
Figure 234 is a figure showing flow header format.
Figure 235 is a figure showing control word formats.
Figure 236 is a figure showing Maximus header format.
Figure 237 is a figure showing ATM header format.
Figure 238 is a figure showing MPLS format.
Figure 239 is a figure showing aal5 trailer format.
Figure 240 is a figure showing system.
Figure 241 is a figure showing reassembly ⇔ memory manager.
Figure 242 is a figure showing reassembly ⇔ SPI.
Figure 243 is a figure showing reassembly ⇔CPU.
Figure 244 is a figure showing internal memory manager to reassembly.
Figure 245 is a figure showing reassembly to spi out - port empty and back to back.
Figure 246 is a figure showing timing diagram-back-to-back port transfer.
Figure 247 is a figure showing memories.
Figure 248 is a figure showing error conditions.
Figure 249 is a figure showing registers.
Figure 250 is a figure showing commands.
Figure 251 is a figure showing read external ras memory.
Figure 252 is a figure showing write external ras memory.
Figure 253 is a figure showing flow control format.
Figure 254 is a figure showing read internal data memory.
Figure 255 is a figure showing write internal data memory.
Figure 256 is a figure showing read four internal memories (partial crc1, partial crc2, reassembly table, output queue).
Figure 257 is a figure showing write four internal memories (partial crc1, partial crc2, reassembly table, output queue).
Figure 258 is a figure showing read port statistics.
Figure 259 is a figure showing write port statistics.
Figure 260 is a figure showing read CPU data memory.
Figure 261 is a figure showing write CPU data memory.
Figure 262 is a figure showing read 2 internal memories (free buffer list (fbl), output control).
Figure 263 is a figure showing write 2 internal memories (free buffer list (fbl), output control).
Figure 264 is a figure showing read internal header data memory.
Figure 265 is a figure showing write internal header data memory.
Figure 266 is a figure showing get CPU cell command.
Figure 267 is a figure showing initialize free buffer list and output port queue memories command.
Figure 268 is a figure showing summary of lookup engine block tags for various traffic types.
Figure 269 is a figure showing ATM header format.

Figure 270 is a figure showing MPLS label format.
Figure 271 is a figure showing ethernet frame format (IEEE 802.3 format).
Figure 272 is a figure showing IP (internet protocol) header.
Figure 273 is a figure showing frame relay (fr) layer 2 frame format (shown with IP payload).
Figure 274 is a figure showing PPP full frame format for unnumbered mode operation.
Figure 275 is a figure showing flow control entry format.
Figure 276 is a figure showing tm to fabric flow control summary.
Figure 277 is a figure showing fabric to tm control summary.
Figure 278 is a figure showing internal block diagram.
Figure 279 is a figure showing graphic representation of an enqueue and dequeue FID.
Figure 280 is a figure showing operation sequence table.
Figure 281 is a figure showing enqueue block diagram.
Figure 282 is a figure showing dequeue block digram.
Figure 283 is a figure showing free buffer manager block diagram.
Figure 284 is a figure showing random number generator.
Figure 285 is a figure showing memory size.
Figure 286 is a figure showing memory chip select.
Figure 287 is a figure showing data dependency.
Figure 288 is a figure showing RED memory.
Figure 289 is a figure showing RED memory byte enable.
Figure 290 is a figure showing statistics counter location 0.
Figure 291 is a figure showing statistics counter location 1.
Figure 292 is a figure showing statistics counter byte enable.
Figure 293 is a figure showing dequeue memory.
Figure 294 is a figure showing dequeue memory byte enable.
Figure 295 is a figure showing FID enqueue location 0.
Figure 296 is a figure showing FID enqueue memory location 1.
Figure 297 is a figure showing FID enqueue memory byte enable.
Figure 298 is a figure showing BID external data structure.
Figure 299 is a figure showing BID external data structure byte enable.
Figure 300 is a figure showing dequeue memory.
Figure 301 is a figure showing dequeue memory byte enable.
Figure 302 is a figure showing multicasting FID enqueue memory location 0.
Figure 303 is a figure showing multicasting FID enqueue memory location 1.
Figure 304 is a figure showing multicasting FID enqueue memory byte enable.
Figure 305 is a figure showing multicasting FID root memory.
Figure 306 is a figure showing FID dequeue leaf tunneling internal data structure.
Figure 307 is a figure showing FID enqueue leaf tunneling internal data structure location 0.
Figure 308 is a figure showing FID enqueue leaf tunneling internal data structure location 1.
Figure 309 is a figure showing FID enqueue leaf tunneling internal data structure byte enable.
Figure 310 is a figure showing tunneling FID root data structure.
Figure 311 is a figure showing interface signals.

Figure 312 is a figure showing CPU interface to PFQ.
Figure 313 is a figure showing segmentation interface diagram.
Figure 314 is a figure showing segmentation interface.
Figure 315 is a figure showing DBS & CBWFQ interface.
Figure 316 is a figure showing DBS interface.
Figure 317 is a figure showing memory and PFQ interface.
Figure 318 is a figure showing memory and PFQ interface signals.
Figure 319 is a figure showing memory manager commands to per flow queue.
Figure 320 is a figure showing per flow queue commands to memory manager.
Figure 321 is a figure showing mem to PFQ commands.
Figure 322 is a figure showing PFQ to mem commands.
Figure 323 is a figure showing memory manager and the per flow queue interface.
Figure 324 is a figure showing PFQ and memory manager PFQ command structure.
Figure 325 is a figure showing PFQ to segmentation timing diagram.
Figure 326 is a figure showing address space.
Figure 327 is a figure showing registers space.
Figure 328 is a figure showing nop command.
Figure 329 is a figure showing read FID enqueue memory command.
Figure 330 is a figure showing write FID enqueue memory command.
Figure 331 is a figure showing read FID dequeue memory command.
Figure 332 is a figure showing write FID dequeue memory command.
Figure 333 is a figure showing read BID memory command.
Figure 334 is a figure showing write BID memory command.
Figure 335 is a figure showing read stat memory command.
Figure 336 is a figure showing write stat memory command.
Figure 337 is a figure showing setup connection command.
Figure 338 is a figure showing tear-down connection command.
Figure 339 is a figure showing init FID memories.
Figure 340 is a figure showing init BID memory.
Figure 341 is a figure showing timing that includes the FID enqueue, dequeue, BID and statistics memories.
Figure 342 is a figure showing enqueue memory.
Figure 343 is a figure showing RED memory.
Figure 345 is a figure showing dequeue memory.
Figure 346 is a figure showing BID memory.
Figure 347 is a figure showing statistics memory.
Figure 348 is a figure showing basic block diagram.
Figure 349 is a figure showing interface to/from shaper.
Figure 350 is a figure showing general signals.
Figure 351 is a figure showing status interface from output spi.
Figure 352 is a figure showing flowid memory.
Figure 353 is a figure showing pkt memory.
Figure 354 is a figure showing port calendar memory.
Figure 355 is a figure showing shp_strict memoy.
Figure 356 is a figure showing shp_out_port memory.
Figure 357 is a figure showing register descriptions.

Figure 358 is a figure showing read external FID memory.
Figure 359 is a figure showing write external FID memory.
Figure 360 is a figure showing setup FID connection.
Figure 361 is a figure showing read external pkt memory.
Figure 362 is a figure showing write external pkt memory.
Figure 363 is a figure showing read port calendar memory.
Figure 364 is a figure showing write port calendar memory.
Figure 365 is a figure showing read shp strict memory.
Figure 366 is a figure showing write shp strict memory.
Figure 367 is a figure showing read shp out memory.
Figure 368 is a figure showing write shp memory.
Figure 369 is a figure showing init shp_out_port memory.
Figure 370 is a figure showing init shp_strict memory.
Figure 371 is a figure showing timing diagram.
Figure 372 is a figure showing shaper.
Figure 373 is a figure showing scheduler.
Figure 374 is a figure showing PFQ.
Figure 375 is a figure showing ingress/egress mode of operations.
Figure 376 is a figure showing interface to/from scheduler.
Figure 377 is a figure showing scheduler's interface.
Figure 378 is a figure showing signal description.
Figure 379 is a figure showing simple round-robin algorithm.
Figure 380 is a figure showing weighted-round-robin scheduling.
Figure 381 is a figure showing creating smaller bursts with factor.
Figure 382 is a figure showing strict priority.
Figure 383 is a figure showing links between all levels of scheduling.
Figure 384 is a figure showing links between memories during input phase (the ingress mode).
Figure 385 is a figure showing links between memories during output phase (the egress mode).
Figure 386 is a figure showing links between memories during output phase (the ingress mode).
Figure 387 is a figure showing external memory – next flowid.
Figure 388 is a figure showing internal memory – qos parameters.
Figure 389 is a figure showing internal memory – qos descriptors.
Figure 390 is a figure showing internal memory – port parameters.
Figure 391 is a figure showing internal memory – port descriptor.
Figure 392 is a figure showing internal memory – next port.
Figure 393 is a figure showing registers.
Figure 394 is a figure showing read FID next memory.
Figure 395 is a figure showing write FID next memory.
Figure 396 is a figure showing read qos parameters.
Figure 397 is a figure showing write qos parameters.
Figure 398 is a figure showing read qos descriptor.
Figure 399 is a figure showing write qos descriptor.
Figure 400 is a figure showing read port parameters.

Figure 401 is a figure showing write port parameters.
Figure 402 is a figure showing read port descriptor.
Figure 403 is a figure showing write port descriptor.
Figure 404 is a figure showing read port next.
Figure 405 is a figure showing write port next.
Figure 406 is a figure showing init qos.
Figure 407 is a figure showing init port.
Figure 408 is a figure showing global timing for input/output stages.
Figure 409 is a figure showing the input phase pipelines.
Figure 410 is a figure showing the output phase pipelines.
Figure 411 is a figure showing shaper/meter block interface.
Figure 412 is a figure showing traffic shaper/meter interface for dual-leaky-bucket.
Figure 413 is a figure showing traffic shaper/meter interface for single-leaky-bucket.
Figure 414 is a figure showing general signals.
Figure 415 is a figure showing progress of a leaky bucket algorithm.
Figure 416 is a figure showing links in the timing wheel.
Figure 417 is a figure showing example for shaping FID in the timing wheel.
Figure 418 is a figure showing calculating future time slot.
Figure 419 is a figure showing the usage of peak and sustained rates.
Figure 420 is a figure showing Rin and K factor.
Figure 421 is a figure showing credit calculation diagram.
Figure 422 is a figure showing calculation of threshold meter.
Figure 423 is a figure showing memory description.
Figure 424 is a figure showing shaper rate ID for shaper mode.
Figure 425 is a figure showing meter threshold for meter mode.
Figure 426 is a figure showing shaper slot memory.
Figure 427 is a figure showing FID mem1 for shaper.
Figure 428 is a figure showing FID mem1 for meter.
Figure 429 is a figure showing FID mem 2 for shaper.
Figure 430 is a figure showing FID mem2 for meter.
Figure 431 is a figure showing register description.
Figure 432 is a figure showing read FID1 memory.
Figure 433 is a figure showing write FID1 memory when using shaper.
Figure 434 is a figure showing read FID2 memory.
Figure 435 is a figure showing write FID2 memory.
Figure 436 is a figure showing read shp_slot memory.
Figure 437 is a figure showing write shp_slot memory.
Figure 438 is a figure showing read rate_id memory.
Figure 439 is a figure showing write rate_id memory.
Figure 440 is a figure showing setup connection command.
Figure 441 is a figure showing read FID1 memory.
Figure 442 is a figure showing write FID1 memory.
Figure 443 is a figure showing read FID2 memory.
Figure 444 is a figure showing write FID2 memory.
Figure 445 is a figure showing read threshold (internal) memory.
Figure 446 is a figure showing write threshold (internal) memory.

Figure 447 is a figure showing read modwrite meter.
Figure 448 is a figure showing setup connection command.
Figure 449 is a figure showing timing diagrams for meter.
Figure 450 is a figure showing block addresses.
Figure 451 is a figure showing read access and timings definitions.
Figure 452 is a figure showing read access timing and definitions table.
Figure 453 is a figure showing write access and timings definitions.
Figure 454 is a figure showing write access and timings definition table.
Figure 455 is a figure showing interface from the CPU IF to the blocks.
Figure 456 is a figure showing configuration modes.
Figure 457 is a figure showing global sync.
Figure 458 is a figure showing slot_count and CPU_port signals.
Figure 459 is a figure showing multiplexing the test busses from the blocks.
Figure 460 is a figure showing interface of the CPU IF block.
Figure 461 is a figure showing internal implementation for locking the current CPU access parameters.
Figure 462 is a figure showing register description.
Figure 463 is a figure showing CPU interface signals.
Figure 464 is a figure showing block addresses.
Figure 465 is a figure showing CPUif block register.
Figure 466 is a figure showing PFQ block register.
Figure 467 is a figure showing read FID enqueue memory (72 bits) - up to 2m locations.
Figure 468 is a figure showing write FID enqueue memory (72 bits) – up to 2m locations.
Figure 469 is a figure showing read FID dequeue memory (36 bits)– up to 1m locations.
Figure 470 is a figure showing write FID dequeue memory (36 bits) – up to 1m locations.
Figure 471 is a figure showing read BID dequeue memory (36 bits) – up to 16m locations.
Figure 472 is a figure showing write BID memory (36 bits) – up to 16m locations.
Figure 473 is a figure showing read stat memory (72 bits) – up to 2m locations.
Figure 474 is a figure showing write stat memory (72 bits) – up to 2m locations.
Figure 475 is a figure showing setup connection command for FID.
Figure 476 is a figure showing tear-down connection command for FID.
Figure 477 is a figure showing init FID memories.
Figure 478 is a figure showing init BID memory.
Figure 479 is a figure showing database block registers.
Figure 480 is a figure showing read external FID memory.
Figure 481 is a figure showing write external FID memory.
Figure 482 is a figure showing setup FID connection.
Figure 483 is a figure showing read external pkt memory.
Figure 484 is a figure showing write external pkt memory.
Figure 485 is a figure showing read port calendar memory.
Figure 486 is a figure showing write port calendar memory.
Figure 487 is a figure showing read shp strict memory.
Figure 488 is a figure showing write shp strict memory.
Figure 489 is a figure showing read shp out memory.

Figure 490 is a figure showing write shp out memory.
Figure 491 is a figure showing init shp_out_port memory.
Figure 492 is a figure showing init shp_strict memory.
Figure 493 is a figure showing shaper/meter block registers.
Figure 494 is a figure showing read FID1 memory.
Figure 495 is a figure showing write FID1 memory when using shaper.
Figure 496 is a figure showing read FID2 memory.
Figure 497 is a figure showing write FID2 memory.
Figure 498 is a figure showing read shp_slot memory.
Figure 499 is a figure showing write shp_slot memory.
Figure 500 is a figure showing read rate_id memory.
Figure 501 is a figure showing write rate_id memory.
Figure 502 is a figure showing setup connection command.
Figure 503 is a figure showing read FID1 memory.
Figure 504 is a figure showing write FID1 memory.
Figure 505 is a figure showing read FID2 memory.
Figure 506 is a figure showing write FID2 memory.
Figure 507 is a figure showing read threshold (internal) memory.
Figure 508 is a figure showing write threshold (internal) memory.
Figure 509 is a figure showing readmod write meter.
Figure 510 is a figure showing setup connection command.
Figure 511 is a figure showing sceduler block registers.
Figure 512 is a figure showing read FID next memory.
Figure 513 is a figure showing write FID next memory.
Figure 514 is a figure showing read qos parameters.
Figure 515 is a figure showing write qos parameters.
Figure 516 is a figure showing read qos descriptor.
Figure 517 is a figure showing write qos descriptor.
Figure 518 is a figure showing read port parameters.
Figure 519 is a figure showing write port parameters.
Figure 520 is a figure showing read port descriptor.
Figure 521 is a figure showing write port descriptor.
Figure 522 is a figure showing read port next.
Figure 523 is a figure showing write port next.
Figure 524 is a figure showing init qos.
Figure 525 is a figure showing init port.
Figure 526 is a figure showing parameters per flowid.
Figure 527 is a figure showing mx2 external memory.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The present invention will now be described in detail with reference to a few preferred embodiments thereof as illustrated in the accompanying drawings. In the following description, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art, that the present invention may be practiced without some or all of these

specific details. In other instances, well known process steps and/or structures have not been described in detail in order to not unnecessarily obscure the present invention.

System Overview

The Maximus implements a multi-service approach with dynamic bandwidth allocation among the various flows carrying the different types of traffic. The Maximus device is designed such that it can be used in different applications that fit the needs of various system architectures. Two of the more popular applications are traffic manager and SAR solutions for OC-192 line cards in routing and switching systems. In one embodiment, the Maximus device is implemented in the form of two integrated circuit chips, designated the MX-1 and MX-2. In another embodiment, the entirety of the Maximus device is implemented on one integrated circuit chip (the circuitry of the MX-1 and MX-2 integrated circuits are integrated onto a single integrated circuit chip).

Figure 1 depicts an exemplary system level overview, illustrating how the Maximus device may enable OC-192 line cards in a router or switch system to receive different types of traffic, to dynamically allocate bandwidth among the various flows, to adapt the incoming traffic to the requirements of the cell-based or packet-based switch fabric, and to convert among the different traffic types for output. The term "router" as it is used in this document includes, but is not limited to, packet routers (for example, IP packet routers) and cell switches (for example, what are commonly called ATM switches).

In Figure 1, each line card can receive both ATM and packet traffic coming in on different flows across a fiber optic connection. The Maximus device on each line card, acting as an ingress device interfacing with a framer device, can receive, buffer and adapt the incoming traffic to the switching requirements of either the cell or packet switch fabric. On the egress side, the traffic can be sent out to the framer device either in the same traffic type (e.g., ATM in, ATM out) or in a different traffic type (e.g., ATM in, packet out or packet in, ATM out). With regard to dynamic bandwidth allocation, the Maximus offers extensive QoS capabilities to shape and schedule the flows as desired according to their QoS priorities.

Figures 2 and 3 below are two exemplary linecard views, illustrating how the Maximus device may be employed on an OC-192 line card operating with either a cell-based switch fabric (Figure 2) or a packet-based switch fabric (Figure 3). In these Figures, the traffic management or SAR solution is accomplished using two separate chips.

In Figure 2, a traffic management solution in conjunction with a cell-based switch fabric is illustrated. Four OC-48 optics couple with a quad OC-48 Framer to achieve an aggregate OC-192 data rate. This is not a requirement, however, as the Maximus Ingress Traffic Manger and the Maximus Egress Traffic Manager can also directly couple with an OC-192 framer to receive data at 10 Gbps. The Maximus supports 64 incoming or outgoing ports and up to 1 million separate flows in each direction. The interface between the Framer and Maximus Traffic Manager chips is SPI-4 (System Packet Interface-4), phase 2 (i.e., LVDS running at 400 MHz). Data storage and buffering employs SDRAMs, while pointers and tables are stored in SSRAMs. The Maximus Traffic Manager chips interface with a cell-based switch fabric in the example of Figure 2.

In the example of Figure 3, the Maximus Ingress Traffic Manger and the Maximus Egress Traffic Manager are employed in the SAR configuration. For example, the Maximus may receive ATM traffic, reassembles the ATM cells into packets along the ingress path to switch through a packet-based switch fabric. On the egress path, the packets may be segmented back into ATM cells for output to the quad OC-48 framers. However, due to the multi-service capabilities offered, packets may also be received and switched through the packet-based switch fabric. The data in these packets (whether received as ATM cells or packets) may be output to the Framer as either ATM cells or packets.

Applications

Figure 4 shows the various applications, along with their combinations of incoming and outgoing traffic types.

Application types 0-3 in Figure 4 represent the ingress traffic manager applications. Application types 8-11, on the other hand, represent the egress traffic manager applications. These ingress and egress traffic management application types (0-3 and 8-11) are graphically represented in Figure 5. Note that traffic management is offered not only for packets but also for ATM cells and AAL5 cells in a single pair of Maximus chips (ingress/egress pair). This reflects the multi-service approach of the Maximus.

Figure 5A shows uses of the Maximus integrated circuit chip with a cell-based switch fabric. The dashed box labeled "MX-1" indicates the contents of one integrated circuit. Certain other parts of the integrated circuit (for example, the shaper block, the scheduler block, and the memory management block) are omitted from Figure 5A in order to clarify the illustration. These other parts are described in further detail below in the later part of this patent document. The darkening of a block in Figure 5A indicates that the block is substantially disabled in that it is not performing its primary function. Blocks that are not darkened in Figure 5A are enabled to perform their primary functions.

Application types 4-6 and 12-14 represent the SAR application types, along with their combinations of incoming and outgoing traffic types. These SAR application types (4-6 and 12-14) are graphically represented in Figure 6. Note that these SAR application types can handle not only segmentation of packets and reassembly of cells but can also support ATM cell encapsulation and packet bypass (application types 4, 6, 12, and 14). Again, this reflects the intelligent multi-service approach of the Maximus.

Figure 6A shows uses of the Maximus integrated circuit chip with a packet-based switch fabric. The dashed box labeled "MX-1" indicates the contents of one integrated circuit. Certain other parts of the integrated circuit (for example, the shaper block, the scheduler block, and the memory management block) are omitted from Figure 6A in order to clarify the illustration. These other parts are described in further detail below in the later part of this patent document. The darkening of a block in Figure 6A indicates that the block is substantially disabled in that it is not performing its primary function. Blocks that are not darkened in Figure 6A are enabled to perform their primary functions.

The data formats for the various application types set forth in Figures 5, 5A, 6 and 6A are discussed below, with reference to the data format drawings of Figures 7-20.

Data Formats

Figure 7 illustrates the data format for Ingress Application Type 0 (ATM => ATM). In this application, the incoming data via the incoming SPI interface is ATM cells (52 bytes, which includes 4 bytes of ATM header and 48 bytes of data). Four bytes of pad is added prior to sending the incoming data to the lookup block (since the Maximus internal data path is 64-bit or 8-byte wide). The segmentation block adds 8 bytes of pad to allow the memory manager to pass the 64-byte memory cells to SDRAMs for storage. The reassembly block adds either 8 or 16 bytes of header to the 64-byte memory cell prior to sending to the cell-based switch (via the outgoing SPI block). The reader is invited to review egress Application type 8 (Figure 14) for the analogous process on the egress side, if the data is to be output to the Framer by the egress Maximus chip as ATM cells. Egress application type 8 is discussed later herein.

Figure 8 illustrates the data format for Ingress Application Type 1 (ATM => MPLS packet). In this application, the incoming data via the incoming SPI interface is ATM cells (52 bytes, which includes 4 bytes of ATM header and 48 bytes of data). Four bytes of pad is added prior to sending the incoming data to the lookup block (since the Maximus internal data path is 64-bit or 8-byte wide). The segmentation block removes the 4-byte ATM header and adds 12 more bytes of pad to allow the memory manager to pass the 64-byte memory cells to SDRAM for storage. The reassembly block adds either 8 or 16 bytes of header to the 64-byte memory cell prior to sending to the cell-based switch (via the outgoing SPI block). The reader is invited to review egress Application type 9 (Figure 15) for the analogous process on the egress side, if the data is to be output to the Framer by the egress Maximus chip as MPLS packets. Egress application type 9 is discussed later herein.

Figure 9 illustrates the data format for Ingress Application Type 2 (MPLS packet =>ATM). In this application, the incoming data via the incoming SPI interface is MPLS packet (Nx16 bytes burst). The packet data is passed onto the lookup engine. The segmentation block segments the packet data into 48-byte AAL5 cells and appends 16 bytes of pad to allow the memory manager to pass the 64-byte memory cells to SDRAM for storage. The reassembly block adds either 8 or 16 bytes of switch header to the 64-byte memory cell prior to sending to the cell-based switch (via the outgoing SPI block). The reader is invited to review Application type 10 (Figure 16) for the analogous process on the egress side, if the data is to be output to the Framer by the egress Maximus chip as ATM cells. Egress application type 10 is discussed later herein.

Figure 10 illustrates the data format for Ingress Application Type 3 (Packet => Packet). In this application, the incoming data via the incoming SPI interface is packet (Nx16 bytes burst). The packet data is passed onto the lookup engine. The segmentation block segments the packet data into AAL5-like 64-byte cells. In this segmentation, CRC-32 is generated and an AAL5-like trailer is added to the end of the last 64-byte cell. The entire 64-byte AAL5-like cell is passed onto the memory manager for storage in SDRAM. The reassembly block adds either 8 or 16 bytes of switch header to the 64-byte memory cell prior to sending to the cell-based switch (via the outgoing SPI block). The reader is invited to review Application type 11 (Figure 17) for the analogous process on the egress side, if the data is to be output to the Framer by the egress Maximus chip as packets. Egress application type 11 is discussed later herein.

Figure 11 illustrates the data format for Ingress Application Type 4 (ATM Encapsulation). In this application, the incoming data via the incoming SPI interface is ATM cells (52 bytes, which includes 4 bytes of ATM header and 48 bytes of data). Four bytes of pad is added prior to sending the incoming data to the lookup block (since the Maximus internal data path is 64-bit or 8-byte wide). The segmentation block adds 8 bytes of pad to allow the memory manager to pass the 64-byte cell to SDRAM for storage. The reassembly block removes 8 bytes of pad, adds either 8 or 16 bytes of switch header prior to sending the resultant 64-byte packet to the packet-based switch (via the outgoing SPI block). The reader is invited to review egress Application type 12 (Figure 18) for the analogous process on the egress side, if the data is to be output to the Framer by the egress Maximus chip as ATM cells. Egress application type 12 is discussed later herein.

Figure 12 illustrates the data format for Ingress Application Type 5 (Reassembly). In this application, the incoming data via the incoming SPI interface is ATM cells (52 bytes, which includes 4 bytes of ATM header and 48 bytes of data). Four bytes of pad is added prior to sending the incoming data to the lookup block (since the Maximus internal data path is 64-bit or 8-byte wide). The lookup block removes the 4-byte ATM header and passes the 48-byte ATM data to the segmentation block, wherein 16 bytes of pad are added to allow the memory manager to pass the 64-byte cells to SDRAM for storage. The reassembly block maps the 48-byte ATM cells stored in memory as 64-byte cell into packets by stripping 16 bytes of pad from each memory cell and adding a switch packet header before sending the resultant packet to the packet-based switch via the outgoing SPI. The reader is invited to review egress Application type 13 (Figure 19) for the analogous process on the egress side, if the data is to be output to the Framer by the egress Maximus chip as ATM cells. Egress application type 13 is discussed later herein.

Figure 13 illustrates the data format for Ingress Application Type 6 (Ingress Packet Bypass). In this application, the incoming data via the incoming SPI interface is a packet burst (Nx16 bytes). The packet data is passed to the lookup block, which subsequently passes the packet data to the segmentation block. The segmentation block segments the packet data into AAL5-like 64-byte cells. In this segmentation, CRC-32 is generated and an AAL5-like trailer is added to the end of the last 64-byte cell. The entire 64-byte AAL5-like cell is passed onto the memory manager for storage in SDRAM. The reassembly block maps the 64-byte AAL5-like memory cell into packets by removing the AAL5 encapsulation and may add 8 or 16 bytes of switch packet header. The reassembled packet, including the 8/16 bytes of switch packet header is passed on to the packet-based switch via the outgoing SPI. The reader is invited to review egress Application type 14 (Figure 20) for the analogous process on the egress side, if the data is to be output to the Framer by the egress Maximus chip as packets. Egress application type 14 is discussed later herein.

Figure 14 illustrates the data format for Egress Application Type 8 (ATM =>ATM). In this application, the incoming data (from the switch) via the incoming SPI interface is a switch cell which comprises 8/16 bytes of switch header, 4 bytes of ATM header, 48 bytes of ATM data, and 12 bytes of pad. The switch header is removed by the lookup block, resulting in a 64-byte cell. The 64-byte cell is then passed through the segmentation block to the memory manager to be stored in SDRAM as a 64-byte memory cell. The reassembly block translates the ATM header, removes 8 bytes of pad (leaving 4

bytes of pad behind to facilitate sending the resultant 52-byte ATM cell + 4 bytes of pad to the outgoing SPI interface (since the Maximus internal data path is 64-bit or 8-byte wide). These 4 bytes of pad are later removed to allow the 52-byte ATM cell to be sent out to the Framer (via the outgoing SPI block).

Figure 15 illustrates the data format for Egress Application Type 9 (ATM =>MPLS packet). In this application, the incoming data (from the switch) via the incoming SPI interface is a switch cell which comprises 8/16 bytes of switch header, 48 bytes of AAL5 data, and 16 bytes of pad. The switch header is removed by the lookup block, resulting in a 64-byte cell. The 64-byte cell is then passed through the segmentation block to the memory manager to be stored in SDRAM as a 64-byte memory cell. The reassembly block strips the 16 bytes of pad, performs a MPLS operation of adding the MPLS packet header based on the contents of the FlowID lookup table before sending the resultant packet out to the Framer (via the outgoing SPI block).

Figure 16 illustrates the data format for the Egress Application Type 10 (MPLS packets =>ATM). In this application, the incoming data (from the switch) via the incoming SPI interface is a switch cell which comprises 8/16 bytes of switch header, 48 bytes of AAL5 data, and 16 bytes of pad. The switch header is removed by the lookup block, resulting in a 64-byte cell. The 64-byte cell is then passed through the segmentation block to the memory manager to be stored in SDRAM as a 64-byte memory cell. The reassembly block adds the 4-byte ATM header and appends 4-byte of pad, as well as strip the 16-byte of pad. These 4 newly added pad bytes are subsequently removed to allow the 52-byte ATM cell to be sent to the Framer (via the outgoing SPI block).

Figure 17 illustrates the data format for the Egress Application Type 11 (packet => packet). In this application, the incoming data (from the switch) via the incoming SPI interface is a switch cell comprising 64 bytes of AAL5-like data and an 8/16-byte switch header. The switch header is removed by the lookup block, resulting in a 64-byte cell. The 64-byte AAL5-like cell is then passed through the segmentation block to the memory manager to be stored in SDRAM as a 64-byte memory cell. The reassembly block maps the 64-byte AAL5-like cells into a packet, and adds a 8/16-byte packet header. The reassembly block also checks the CRC in the AAL5 trailer before removing the AAL5-like encapsulation. The resultant packet is subsequently passed on to the Framer (via the outgoing SPI block).

Figure 18 illustrates the data format for the Egress Application Type 12 (ATM de-encapsulation). In this application, the incoming data via the incoming SPI interface is a switch packet, which comprises 8/16 bytes of switch header, 4 bytes of ATM header, 48 bytes of ATM data, and 4 bytes of pad. The switch header is removed by the lookup block, resulting in 56-byte cell. The 56-byte cell is passed to the segmentation block, which adds 8 bytes of pad to allow the memory manager to pass the 64-byte cell to SDRAM for storage. The reassembly block translates the ATM header, strips out 8 bytes of pad (leaving 4 bytes to allow passing through the 8-byte/64-bit data path from the reassembly block to the outgoing SPI block). These last four bytes of pad will be removed, allowing the outgoing SPI to send the 52-byte ATM cell to the Framer.

Figure 19 illustrates the data format for the Egress Application Type 13 (Segmentation). In this application, the incoming data via the incoming SPI interface is a packet burst, including the 8/16 bytes of switch header. The packet data is passed onto

the segmentation block via the lookup block. At the segmentation block, the packet data is segmented into 48-byte AAL5 cells, and 16 bytes of pad are added to allow the memory manager to pass the 64-byte cells to SDRAM for storage. The reassembly block removes the 16 bytes of pad, adds the 4-byte ATM header to the 48-byte ATM data, and pads with 4 additional bytes to allow passing through the 8-byte/64-bit data path from the reassembly block to the outgoing SPI block. These last four bytes of pad will be removed subsequently, allowing the outgoing SPI to send the 52-byte AAL5 cell to the Framer.

Figure 20 illustrates the data format for the Egress Application Type 14 (Egress Packet Bypass). In this application, the incoming data via the incoming SPI interface is a packet burst, which comprises 8/16 bytes of switch header and 64 bytes of AAL5 data. The packet data is passed onto the segmentation block via the lookup block. At the segmentation block, the packet data is segmented into 64-byte AAL5-like cells for storage in SDRAM memory. The reassembly block maps the 64-byte AAL5-like cells into packet, and adds a packet header of 0-16 bytes. It also checks the CRC in the AAL5 trailer before removing the AAL5 encapsulation. The packet is then passed to the Framer via the outgoing SPI block.

Block Diagram

Figure 21 is a high-level block diagram of the Maximus 1 integrated circuit chip, the Maximus 2 integrated circuit chip, and various memory devices coupled to the Maximus 1 and 2 integrated circuit chips. All of these integrated circuits are present, in one embodiment, on a line card of a router. The functionality of the Maximus 1 (MX-1) and Maximus 2 (MX-2) integrated circuit chips may be disposed on two different integrated circuit chips as illustrated in Figure 21. Alternatively, the circuitry of the Maximus 1 and Maximus 2 integrated circuit chips is combined such that all that circuitry is disposed on a single integrated circuit chip.

Features

General Features

- Supports full duplex 10Gbps performance
- Protocols and internetworking
  - ATM
    - Operation and Management (OAM) cells
    - Signaling support
    - VP/VC translation
    - ATM to packet internetworking with Segmentation and reassembly
    - VC merging support
  - Packets
    - MPLS
      - MPLS tag switching
      - IP to ATM internetworking with AAL5 segmentation and re-assembly.
      - MPLS header generation. Used for MPLS tunneling and aggregation

- MPLS merging several flows from different ports merged to single output.
- Any packet with MPLS-like header

- Management support
  - SNMP, MIB counters
  - Alarms and Events
- Multicast support for both ATM and Packets

- Applications supported:
  - Ingress Traffic Manager
    - ATM Cells in, ATM Cells out.
    - ATM (AAL5) Cells in, packets out.
    - MPLS Packets in, ATM (AAL5) Cells out.
    - MPLS Packets in, MPLS Packets out.
  - Egress Traffic Manager
    - ATM Cells in, ATM Cells out.
    - ATM (AAL5) Cells in, Packets out.
    - MPLS Packets in, ATM (AAL5) Cells out.
    - MPLS Packets in, MPLS Packets out.
  - Reassembler
    - ATM (AAL5) cells in, packets out
    - ATM cells in, ATM cell encapsulated inside packet out.
    - MPLS Packets in, MPLS Packets out.
  - Segmentation
    - MPLS Packets in, ATM (AAL5) Cells out.
    - ATM cell encapsulated inside packet in, de-encapsulated ATM cell out.
    - MPLS Packets in, MPLS Packets out.
- Interfaces supported
  - Generic CPU Interface
  - SPI-4 phase-2 interfaces for both packets and cells.
    - 16-bit, 400 MHz, DDR interface.
    - Supports up to 64 ports (any combination of STS-1 – STS-192).
    - SPI-4 used for both incoming and outgoing interfaces.
- Traffic management and QoS mechanisms
  - Traffic Management on Ingress
    - Priority scheduling with weighted queuing
      - Per Flow per egress port per class scheduling (up to 1k queues)
      - Flow control with the switch
  - Traffic Management on Egress
    - Traffic shaping through dual leaky bucket
    - Traffic scheduling
      - 64 Utopia level 4 ports
      - 8 Class of service
      - GCRA scheduling

- Weighted fair queuing
  - VP tunneling support
  - CBR tunneling
  - MPLS tunneling

CPU Interface

- Supports CPU interface for management and control information.
- Supports 32-bit CPU bus interface.
- Support Mux/Non-Mux and Big Endian/Little Endian modes
- Supports interrupt generation.
  - Hierarchical interrupt structure

Incoming SPI-4 Interface

- Supports the SPI-4 phase II specifications for 10 Gbps rate.
- Unpacks incoming 16-bit DDR data at 400 MHz into 64-bit wide word at 200 MHz.
- Performs diagonal interleaved parity checking.
- Extracts in-band port address, SOP, EOP and error-control code.
- In-band start-of-FIFO status signal.
- 200 MHz for the FIFO status transfer.
- Supports maximum 64 ports.
- Generates 2-bit FIFO status back to the transmit device using programmed port calendar.
  - Programmable port calendar with STS-1 granularity.

Lookup Engine

- Supports up to 1M flows.
- Supports up to 64 input ports.
- Tables and configuration registers loaded via the processor interface.
- Lookup Algorithm
  - Label generator selects any 6 bytes from the first 32 bytes of the packet.
  - Label generator is programmable on a per port basis.
  - The label generator has one byte of granularity.
  - Hash generator creates 2 hash values or slots in parallel from a key using 2 separate CRC polynomials as hash functions.
  - Four bins per bucket to minimize collisions.
  - Hash table lookup constrained to resolve within 4 tries.
  - Overflow CAM of 256 or 64 entries, the largest available, for lookups that fail to resolve in 4 tries.
  - Fully pipelined, one lookup every 8 cycles.
- Supports the following flows with the following headers:
  - MPLS with PPP encapsulation.
  - MPLS with LLC/SNAP encapsulation.
  - MPLS with Frame Relay encapsulation..
  - Custom switch headers for traffic manager application.
  - Supports bypass lookup using MPLS-like header.

Segmentation Engine

- Segments incoming packet data into cells on a per port basis

- Supports a maximum of 64 external ports
- Supports an additional port for data from processor
- Segments incoming packet data into ATM cells
- Appends AAL-5 trailer for ATM segmentation
- Segments incoming packet data into Switch cells of 64 bytes
- Appends AAL-5 like trailer for non-ATM segmentation

- Supports programmable maximum packet length of 16K bytes.
- Gathers per port statistics
  - 32-bit counter for incoming packets.
  - 48-bit counter for byte count.
  - 16-bit counter to count discarded packets.
- Tags bad packets from SPI, to be discarded by per flow queue.
- Internal SRAM to store per port segmentation context and data

Memory Manager

- Uses two channels of 128-bit (64-bit DDR) SDRAM
- Manages read and write requests to all eight banks separately to maximize utilization.
- Uses 128kb of internal SRAM for temporary storage of data to and from SDRAM.
- Sopisticated load-sharing algorithm to distribute traffic over various banks.

Reassembly and Header Processing Engine

- Types of operation:
  - ATM cells to ATM cells no action required with translation
  - ATM cells to ATM cells with translation
  - ATM cells to Switch cells
  - Switch cells to Switch cells
  - Switch cells to Packet Data
  - ATM cells to Packet Data
  - AAL-5 cells to ATM cells, appending ATM header
  - AAL-5 cells to Packet Data

Header Processing Features

- Supports a maximum of 1M total flows.
- Types of lookup supported
  - No lookup, pass-through traffic
  - ATM processing
    - ATM Header append
    - VPI, VCI header translation
    - Encapsulate raw cells and OAM cells into packets for routing through packet fabrics.
    - Encapsulate ATM cells into packets for routing through packet fabric.
  - MPLS processing
    - MPLS push/pop
    - MPLS header translation

Reassembly Features

- Reassembles incoming cells into packets on a per port basis
  - Supports a maximum of 64 external ports
  - Supports an additional port for data to processor
  - Reassembles incoming ATM cells into packet data.
  - Strips AAL-5 trailer for ATM reassembly
  - Segments incoming Switch cells of 64 bytes into packet data.
  - Strips AAL-5 like trailer for non-ATM reassembly.
- Supports programmable maximum packet length of 16K bytes.
- Gathers per port statistics
  - 32-bit counter for outgoing packets.
  - 48-bit counter for byte count.
  - 16-bit counter to count discarded packets.
- Tags bad packets to SPI, to be discarded by downstream device.
- Internal SRAM to store per port segmentation context and data

Outgoing SPI-4 Interface

- Supports SPI-4 phase II spec at 10 Gbps rate.
- Packs the 64-bit data at 200 MHz to 16 bit DDR data at 400 MHz.
- Multiplexes the data and the control onto the SPI-4 bus.
- Diagonal Interleaved parity generation.
- Programmable data transfer burst sizes of multiple 16 bytes.
- Passes the FIFO status information to the scheduler to stop transferring.
- Schedules training sequence for data path de-skew.
- Supports maximum 64 ports
- Receives 2-bit FIFO status from the receive device using programmed port calendar.
  - Programmable port calendar with STS-1 granularity.

Per Flow Queue Engine

- See below, page 88.

Traffic Shaper and Traffic Meter

- See below, page 129.

Output Scheduler

- See below, page 118.

Signals

See Figure 22.

MX 1 CPU Interface

This block is responsible for three different functions:

- Connects between the blocks and the CPU via the CPU interface. It decodes the address to Figure which block to access and generate the chip select to each block.
- Generates global synchronization signals to the rest of the blocks.
- Holds general control registers like version register, mode register etc.

Features

1. Address bus of 10 bits per chip.
   Bits 0-5 are used as an address inside the block (up to 64 registers per block)
   Bits 9-6 are used to select a block (up to 16 blocks per chip)
2. Data bus of 32 bits per chip.
   Externally this bus is b-directional
   Internally this bus is split to 2 busses: DATA_IN and DATA_OUT.
3. Chip select signal is the qualifier for data, address and read/write busses.
4. The CPU interface can support multiplexed CPU interface or non-multiplexed CPU interface. When using multiplexed interface, the address and data are folded to the DATA[31:0] bus.
5. CPU accesses all registers in all blocks via direct read/write commands.
6. CPU accesses all internal/external memories in all blocks via indirect accesses using the general-purpose registers dedicated for these accesses.
7. Blocks addresses can be found in Figure 23.
8. Generates a global sync signal once every 520 cycles of the 200MHz clock (65 ports, 8 cycles per port).

Functional Description

The CPU IF block is used to interface with the CPU. The CPU IF block decodes the incoming address bus to generate a unique chip-select signal to each block. The CPU IF block also holds general registers that contain general parameters and modes for the entire chip. Basically any logic that is not related directly to a specific block is in this block.

The CPU IF block can operate in two modes depending on an input pins to select between them:

- Regular CPU access: separate busses for data and for address. In this case address bus is qualified be the chip-select signal.
- Multiplexed CPU access: one bus is multiplexed for address and for data. In this case address is qualified by the ALE signal.

The CPU interface is based on Intel's microprocessor interface (when multiplexing address/data bus). The following timing diagrams defines the characteristics of the CPU interface for Read and for Write accesses:

Read access characteristics

See Figures 24 and 25.

- RDWR_L signal has the same timings as the ADDR bus.
- A valid read cycle is defined when RDWR_L is latched as high on the negative edge of the CS_L signal.
- In non-multiplexed address/data bus architecture, ALE should be held high so parameters t_{Salr}, t_{Halr}, t_{VL}, t_{Slr} and t_{Hlr} are not applicable.
- In our implementation the address is latched on the negative edge of the CS_L therefore t_{Har} is not applicable.

Write access characteristics

See Figures 26 and 27.

- RDWR_L signal has the same timings as the ADDR bus.
- A valid write cycle is defined when RDWR_L is latched as low on the negative edge of the CS_L signal.
- In non-multiplexed address/data bus architecture, ALE should be held high so parameters t_{Salw}, t_{Halw}, t_{VL}, t_{Slw} and t_{Hlw} are not applicable.
- In our implementation the address is latched on the negative edge of the CS_L therefore t_{Haw} is not applicable.

Timing diagram of signals to blocks

The diagram in Figure 28 describes the timings of the CPU interface to each block. Note that chip-select signal is now asserted for only one cycle of the 200MHz clock. For a write or read cycle, all parameters are valid and stable before chip-select is asserted. When a read cycle occurs data is driven from the block after a maximum of 3 clock cycles of 200MHz clock.

GSYNC - Synchronization signal

GSYNC signal is a one-cycle pulse in the 200MHz clock, which is asserted once every 520 cycles. The SYNC signal is used for synchronization between the blocks and between the chips as well as to generate the slot-count and the CPU-port indication. See Figure 29.

Slot count and CPU port indication

The signals slot_count and CPU_port are generated inside each block base on the GSYNC signal. Each block first needs to sample GSYNC and use the sampled version to generate the signals. See Figure 30.

Interface

The CPU IF block generates chip-select signal per block, the rdwr_l, addr and data busses are propagated to the blocks directly. The blocks perform the read or write command based on the rdwr_l signal only when the chip-select signal per the block is asserted. The chip-select signal per each block is synchronous to the 200MHz clock domain already. Each block first samples the incoming signals and then uses them. This is being done to solve any timing issues that may occur. If during placement and routing the signals from the CPU IF to the blocks will violate the 5nSec budget, another set of FFs can be placed in the middle of the path.

For a read cycle the CPU IF block selects the data busses coming in from the blocks based on the chip-select per block (the address). See Figure 31.

The address, data and rd/wr signals takes about 4 cycles of 200MHz to propagate from the chip's pins to the block (6 cycles are used for delay calculation). 2 cycles are needed to perform read or write accesses in the block.

Therefore a total of 8 cycles are needed to perform a read or a write access.

Block Descriptions

CPU Interface

The negative edge of the CS_L signal qualifies the ADDR[9:0] and the RDWR_L signals. The positive edge of the CS_L signal qualifies the DATA bus for read and for write accesses. The CS_L is basically an enable signal to latches. The circuit shown in Figure 32 shows the basic latching circuit and the synchronization to 200MHz clock domain:

Test Mux

The Test Mux block is used to multiplex test mux outputs from all the blocks inside the chip. The output bus of the test mux go out to the pins that will be used for testing and debugging purposes. Refer to the Figure below for the Test Mux block diagram. The Tstmux_Blk_Sel register (refer to the Register section) is used to select which block signals to be output to the test pins (Tstmux_pin[31:0]). Each block inside the chip also has selection register (e.g. Seg_Tstmux_Sel, Ras_Tstmux_Sel) to select a group of internal signals to be observed.

The test mux outputs of all the blocks should be clocked out of flip-flops (refer to the diagram below). The CPU_clk that is running at 200 Mhz, is used to clock the output of the test mux inside the CPU Interface block. It is also going out to a test pin called Tstmux_clk. See Figure 33.

Soft Reset

Each block inside MX-1 chip can be reset individually by writing a '1' to its Soft_Reset register (refer to the Register section). Writing a '0' to the register will release the block from being reset and get it back to its normal operation. Each soft reset signal of each block is also coupled with the global power-on reset pin (POR_L). Refer to Figure 34 below.

Input/Output Disable

Each block in the MX-1 chip has 2 inputs coming from the CPU Interface block that are used to disable the input and output of the block. The Input_Disable and Output_Disable registers (refer to the Register section) can be programmed to disable the input or/and output of a particular block. Refer to Figure 35 below.

Registers

The following table describes the registers implemented in the CPU IF block. These registers act as control to the rest of the chip. The registers are implemented in the 200MHz clock domain even if there is no need to do so since their value is fixed and stable during normal operation. See Figure 36.

Incoming SPI-4

The Incoming System Packet Interface Level 4 (SPI-4) block receives 16-bit DDR ATM or Packet over SONET/SDH (POS) data from the Transmit Device (PHY or Switch based on ingress or egress application) and sends out-of-band FIFO status information to the Transmit Device. The incoming SPI-4 interface block outputs data

with control information to the lookup engine, and receives FIFO status from segmentation engine. Furthermore, the CPU can program the registers in the incoming SPI-4 interface block via CPU interface.

Features

- Supports the SPI-4 phases II spec at 10 Gbps rate
- Packs incoming 16-bit DDR data at 400MHz into 64-bit wide word at 200MHz
- Free buffer list to manage per port FIFO and absorb data while look up engine is doing the look up
- Diagonal interleaved parity checking for data
- Extracts in-band port address, SOP, EOP and error-control code and presents to the lookup engine
- Passes 2-bit FIFO status back to the transmit device in a round robin fashion based on the port calendar
- Programmable port calendar to output FIFO status
- In-band start-of-FIFO status signal
- 400 MHz for the FIFO status transfer
- De-skews +/- 1 bit time after running the training patterns
- Supports maximum 64 ports

Functional Description

The incoming SPI-4 interface block receives data with in-band port address, delineation information and error control code from an external transmit device. It separates the data and control and passes them to the lookup engine. It also sends out-of-band flow control information to the transmit device when it receives the information from the segmentation block. See Figures 37, 38 and 39.

Block Description

Bit Align

The bit-align block adds programmable delays to both data and control lines to correct for relative skew differences of up to +/- 1 bit time. The programmable delays are controlled by the bit align control block which receives pre-defined training patterns to determine the adjustment. It outputs 32 bit SDR data with two control signals. The control signals indicate which 16-bit word is data or control word. See Figure 40.

Bit Align Control

This block compares the received training patterns with the pre-defined patterns and determines the receive timing of the lines. Based on the timing, it generates the control signal the select different delays to de-skew the data and control lines.

Drop IDLE

This block drops the extra idle control words from the incoming data and writes only data, training pattern, and the first detected idle control word into the input FIFO. This block has a state machine which powers up in the idle state and remains in that state

until data or the first idle control word is detected. For the first detected idle control word, the state machine advances to the state named FIRST IDLE; and for non-idle control word it advances to the state named SCAN. If the state machine is in the FIRST IDLE state, it advances to the SCAN state if non-idle control word is detected, otherwise, it goes back to the IDLE state and asserts first_idle signal. If the state machine is in the SCAN state, it stays in this state until idle control word is detected and it then advances to the FIRST IDLE state. The incoming data and control words are allowed to pass through and written together with the control signals into the input FIFO when the state machine is in the SCAN or FIRST IDLE states. See Figure 41.

Input FIFO

This input FIFO is a 256x34 dual-port SRAM running at 400MHz to absorb incoming SPI-4 data, control words, and control signals. The write operation is controlled by the drop IDLE state machine, and the read operation is controlled by the receive state machine (RX_SM) when the FIFO is not empty.

Rx State Machine

The Rx state machine receives 32-bit words from the input FIFO with indications of data or control words of the two 16-bit words. It extracts the in-band control information, packs the data into 64-bit wide words, passes the training patterns to the bit align control block. It also sends the data and the control words to the parity-checking block to check the parity.

A common control word format is used in both the incoming and outgoing interfaces. The Figure below describes the fields of the control word. When inserted in the data path, the control word is aligned such that its MSB is sent on the MSB of the incoming or outgoing data lines. A (payload) control word that separate two adjacent burst transfers may contain status information pertaining to the previous transfer, the following transfer, or both transfers. The table below shows a list of control word types. Maximus would ignore reserved control when detected from the incoming SPI-4 interface.

Figures 42 shows the SPI-4 control word format and Figure 43 and 44 contain the detailed description for the SPI-4 control word format.

The Rx state transition table is shown in Figure 45 and is based on the types of control words received. Since the state machine has to decode two 16-bit words in one cycle, the table lists all the possible 16-bit words combinations. The 2-bit **ctrl** signal is derived from the input DDR signal RCTL. Bit 1 is for the word 1 and bit 0 for word 0. 1 indicates a control word and 0 is a data word.

I_ctrl = Idle control word detected P_ctrl = Payload control word detected

T_ctrl = Training control word detected

Parity Checking

The parity-checking block calculates 4-bit Diagonal Interleaved Parity (DIP-4) and compares the calculated DIP-4 with the one received in the control word. Then it reports the error. The extent of the parity calculation is shown in Figure 46. A functional description of calculating the DIP-4 code is given in Figure 47.

Sync FIFO

This FIFO is an 8x64 dual-port SRAM used to synchronize the 200MHz data/controls from the incoming SPI-4 to the internal clock. The write operation is controlled by the RX_SM and the read operation is controlled by the per_port control block. The per_port control reads from the sync FIFO as long as it is not empty.

SPI TBL

The SPI TBL is a table of 64 entries, one for each port. The table contains control information for the incoming data burst for each port.

FBL

The FBL (Free Buffer List) block is an 84x10 single-port SRAM containing the pointers to the data FIFO that are free to accept data.

Data FIFO

The data FIFO is a 672x64 dual-port SRAM consists of 84 blocks of 64 bytes. It is used to receive data for each incoming port and only send the data out if it has accumulated 64 bytes or reached end of packet.

Calendar

The calendar block contains a port calendar conFigured at the startup time. An example of a port calendar is shown in Figure 48.

The calendar block receives per-port-basis FIFO status from the segmentation and encodes the status of each port in a 2-bit data structure. Then it sends the status back to the transmit device in a round robin fashion based on the port calendar. The "11" pattern is reserved for in-band framing. A DIP-2 odd parity is sent at the end of each complete sequence, immediately before the "11" framing pattern. See Figure 49. A description of calculating DIP-2 is shown in Figure 50.

Memory Description

Summary of Internal Memories:

See Figure 51.

Interface

Interface Ports

See Figure 52.

Register Description

See Figure 53.

Principles of Operation

The incoming SPI block receives 16-bit DDR data, 1-bit DDR control and a 400 MHz clock from the LVDS pins of the Maximus chip. It first converts the 16-bit DDR data words into 32-bit SDR data words and a 1-bit DDR control into a 2-bit SDR control. The 2-bit control indicates which 16-bit word within a 32-bit word is the control word or data word. Based on the 2-bit control, the Rx state machine starts to extract the in-band

port address, start/end-of-packet indication and error-control code from the 32-bit data path and packs the 16-bit data words into 64-bit data words.

Subsequently, the incoming SPI block outputs the 64-bit data words along with the control information at 200 MHz to the lookup engine. The incoming SPI block also calculates 4-bit Diagonal Interleaved Parity (DIP-4) and reports an error if the calculated DIP-4 is not equal to the one received with the control word.

The incoming SPI block also sends 2-bit FIFO status information back to the transmitting device. This FIFO status information is originated from the segmentation block indicating FIFO congestion conditions for all the ports. The SPI block sends the FIFO status to the transmitting device at the rate of one port per cycle in a round robin fashion based on the port calendar sequence.

Outgoing SPI-4 Interface

The Outgoing System Packet Interface Level 4 (SPI-4) interface transmits 16-bit DDR ATM or Packet over SONET/SDH (POS) data to the receive device (PHY/Switch, depending on whether employed in the ingress or egress application) and receives out of band FIFO status information from the receive device. The outgoing SPI interface gets data and control information from the re-assembly engine, and passes the FIFO status information to the database. Furthermore, the outgoing SPI-4 interface also interfaces with the CPU through a software interface. The CPU can program the registers inside the blocks that comprise the Outgoing SPI-4 Interface.

Features

- Supports SPI-4 phase II spec at 10 Gbps rate.
- Unpacks the 64-bit data at 200 MHz to 16 bit DDR data at 400 MHz.
- Muxes the data and the control onto the SPI-4 bus.
- Diagonal Interleaved parity generation.
- Programmable data transfer burst sizes of multiple 16 bytes with minimum of 80 bytes.
- Passes the FIFO status info to the database to stop transferring.
- Schedules training sequence for data path de-skew.
- Supports maximum 64 ports.

Functional Description

The Outgoing SPI-4 Interface receives 64-bit data and control information from the re-assembly engine at 200 MHz. The Tx state machine generates 16-bit control words, unpacks the 64-bit data into 16-bit data words, and multiplexes the control and data words onto a 16-bit, 400 MHz DDR bus to an external receive device. The Outgoing SPI-4 Interface also passes the flow control information from the receive device to the database block within the Maximus chip. See Figure 54.

Block Description

Tx State Machine

The Tx state machine stores and monitors the length field from the reassembly block. The length field is decremented on every data transfer from reassembly to SPI.

When the length reaches a threshold, the Tx state machine issues a request command to read the next port data and control information out of reassembly block. The threshold level is determined by a value that would allow back-to-back data transfer from reassembly block into SPI without gap on the data bus.

The control word format is common for both incoming and outgoing interfaces. Refer to the Incoming SPI Interface document for details.

The state machine also schedules to send the training patterns for the data path de-skew at least once every pre-defined DATA_MAX_T cycles. The training pattern is shown in Figure 55 from cycles 1 through $20\alpha + 1$. The α is the number of repetitions of the data training sequence. The DATA_MAX_T and α are conFigured at the startup.

The state transition table is shown in Figure 56. At each state, two 16-bit words and pos_v and neg_v have to be output. And the pos_v and neg_v will be used to generate a DDR control signal. 1 indicates a control word and 0 a data word. If data_valid is set, there are data to be sent. Otherwise the state machine needs to generate one of the control words.

Data Parity Generation

This block calculates 4-bit Diagonal Interleaved parity (DIP-4) and sends the DIP-4 along the control words. The extent of the parity calculation is shown in Figure 57. A functional description of calculating the DIP-4 code is given in Figure 58.

Unpack Engine

This block receives 64 bit data from Reassembly block and samples data at 200MHz clock rate into the SPI out block. See Figure 59.

The 64-bit data and a data valid signal are received following the control info read. The unpack engine unpacks the 64-bit input data into 16-bit words. In order to send 16-bit DDR data at 400 MHz to the LVDS I/Os, the state machine has to come up with two 16-bit words every 400 MHz clock cycle. See Figure 60.

Configuration Register

This block defines the organization of configuration space based on SPI-4 phase 2 specifications.

Port Calendar

This block stores the predefined calendar information to identify each port is ready to send data via the SPI interface. The port calendar contains three fields to indicate the PORT ID, JUMP and VALID. The port ID represents the port number for transmission of data and receiving of status. The port calendar can store up to 64 ports, in the event that less that 64 ports are used, JUMP indicates the end of calendar. VALID indicates if the port is valid or not. If a given port is not valid, the next PORT ID should be used, and the PORT ID marked as not valid should be skipped.

When CPU updates the port calendar after initialization, CPU needs to meet the following conditions:

- CPU can only modify one port at a time.
- When CPU write a port, CPU needs to specify a port entry in the port calendar between port 0 to 63, and the specific port will be modified.

- When CPU read a port, CPU needs to specify a port entry in the port calendar between port 0 to 63, and the specific port will be read out.
- CPU needs to stop the traffic for a given prior to modifying the port attribute (Jump, Valid).

See Figure 61.

Port Calendar State Machine

This block waits for rdy_4_nx_port_id from TX state machine to indicate the transmit state machine is ready to accept another port request. Upon receiving rdy_4_nx_port_id, calendar state machine issues the next port ID from the port calendar to the TX state machine. However, before issuing the next port ID, the status bit is check to ensure the SPI FIFO status is in the "Hungry" state for the given port. There is a 64-bit register to retain the value status on a per port basis. The block matches incoming status arriving into SPI against attributes from Port Calendar to identify the given port traffic status.

After Port Calendar issues a valid port to the TX state machine, the Port Calendar State Machine pre-fetches the next Port ID from Port Calendar by performing a read. If the TX State Machine issues a read before a valid Port ID is inside Port Calendar State Machine, no_valid_port_id_available is returned to the TX State Machine.

Status Parity Check

This block checks 2-bit Diagonal Interleaved parity (DIP-2) and verifies DIP-2 parity bits for signal integrity. Assuming the TSCLK is running at 400 MHz, this block operates at the 400 MHz clock domain and samples the tstat [1:0] twice every clocks.

There are two 65-bit registers to hold the FIFO status. The first 65-bit registers hold the status value before parity check (pre_chk_st) and the second register holds the status after parity check (post_chk_st). Once the parity check passes, this block is responsible to copy the status from the pre_chk_st to the post_chk_st register. If there is a parity error, the error is reported into the control status register (spio_err [0]) by setting 1'b1 into the bit. This spio_err register is self-cleared upon CPU read.

This block sends SPI status from post_chk_st to the Database block. The SPI port status will be sent in spio_dbs_stt [7:0], and status for port 0-7 should be sent when the spio_dbs_sync is asserted, and status for port 8-15 should be sent in the following clock, status 16-23 on the next clock, etc. SPI status spio_dbs_stt [8] should contain the status for the CPU port. spio_dbs_sync should be derived from the global_sync signal.

When tstat [1:0] is 2'b01, this indicates "Hungry" and transfer of up to ((MaxBurst2)*(16-byte)) blocks or the remainder of what was previously granted (whichever is greater) may be sent. Since Maximus internal cell size is 64-byte plus header (up to 16 bytes), after SPI-out provides an indication to Maximus to stop scheduling additional cell to memory, SPI-out needs to allow the remainder of current cell to go out of SPI. In other words, reassembly block does not support shipment of partial cell to SPI, each time SPI request for a cell, SPI must guarantee to be able to accept one complete cell. Therefore, the MaxBurst2 value should be programmed to minimum of 5. Once programmed to 5, when the "Hungry" indication is received, the remaining bytes in a given cell can still be shipped out. To notify Maximus to stop scheduling the next cell of data is the spio_dbs_stt bit from SPI-out to Database block.

Consequently, spio_dbs_stt should be set to one when "Hungry" is received for a given port.

This interface between SPI-out and database drives the status of the 64 ports in the output SPI and the CPU port. It is a continuous interface. Every 8 cycles of 200MHz the status indication of the 65 ports is updated to a 65 bits register. The first cycle of the eight is marked with a specific signal (sync signal).

Figure 62 shows the interface:

A synchroniztion FIFO is used between the 400 MHz and 200 MHz clock domain.

This block detects the synchronization pattern from tstat [1:0] by comparing the incoming tstat[1:0] against the framing pattern. Also, the port calendar state machine checks training sequence to see if there is any skewing on FIFO status and control lines. To check the training sequence for skew, this block needs to implement a counter for internal tstat, called sync [1:0], which cycles the nominal value of tstat [1:0] with respect to Maximus internal clock. The value of sync is compared against the value of tstate [1:0] (external) from the SPI pad and the difference between the two pairs of signals determines if there is any skew of control signals. See Figure 63.

TX FIFO

This block implements a 4x16-bit dual port SRAM to synchronize Maximus and SPI clock domains. On the incoming side, the write port operates at 200 MHz and pushes data into the TXFIFO when data_push is asserted by TX state machine. On the outgoing side, the read port operates at 400 MHz DDR and pops data into the tdat [15:0] when data_pop is asserted by TX state machine.

TX Status FIFO

This block synchronizes status from 400 MHz to 200MHz domain using 4x16-bit dual port SRAM.

Memory Description

Data Structure for Port Calendar

See Figure 64.

- Memory type: single-port SRAM
- Memory configuration: 64 x 8 bit
  - Port ID: hold port identification value programmed by the CPU.
  - Jump: indicates the end of the port calendar. When Jump is asserted, the given location is the last entry on the port calendar. Upon detecting the assertion of jump, the port calendar should recycle the port ID from the top of the port calendar.
  - Valid: When valid is asserted, it indicates the given port hold a valid connection. When valid is de-asserted, the port calendar will skip the de-asserted port and go to the next port ID on the port calendar.

Data Structure for TX FIFO

- Memory type: dual-port SRAM

- Memory configuration: 4 x 16 bit

The TX FIFO is used to synchronize data between different clock frequencies.

Data Structure for Status FIFO

- Memory type: dual-port SRAM
- Memory configuration: 4 x 16 bit

The Status FIFO is used to synchronize data between different clock frequencies.

Interface

See Figure 65.

Timing Diagram

Timing information between SPI-out and Reassembly, see Figures 66 and 67.

Register Description

Start-up Parameters, see Figure 68.

Software Interface

The software interface programs Outgoing SPI with a CPU. All Outgoing SPI's control registers and memories can also be written and read for test purposes through the software interface. The following sections list all the control registers and all the Outgoing SPI software interface commands. Please see a dedicated document for a detailed description of the software interface.

Registers

See Figure 69.

Commands

Read Outgoing SPI Start-up Parameters.

See Figure 70.

Write Outgoing SPI Start-up Parameters

See Figure 71.

Read Outgoing SPI Start-up Parameters

See Figure 72.

Write Outgoing SPI Start-up Parameters

See Figure 73.

Read Outgoing SPI Start-up Parameters

See Figure 74.

Write Outgoing SPI Start-up Parameters

See Figure 75.

Write PORT Parameters after Initialization

See Figure 76.

Read PORT Parameters after Initialization

See Figure 77.

Read PORT Full Parameters

See Figure 78.

Write PORT Full Parameters

See Figure 79.

Read Control Status Register

See Figure 80.

Write SPIO_TSTMUX_SEL

See Figure 81.

Read SPIO_TSTMUX_SEL

See Figure 82.

Reserved Op-codes

All op-codes other than listed above are reserved.

Principle of Operation

The outgoing SPI block receives FIFO status from the Incoming SPI. The beginning of the status is found based on the framing sequence. The beginning of the status from SPI interface is mapped to the port calendar to look up the status of each port. The status has to be synchronized to the internal 200 MHz clock and passed to the database block, and internal state machine to select a port that is available for data

transmission. Base on the status information, outgoing SPI block requests data from re-assembly block on a given port. Outgoing SPI unpacks the port data from re-assembly, delineation data between different ports and check for data error using control information from re-assembly. The outgoing SPI block calculates the 4-bit Diagonal Interleaved Parity (DIP-4) and combines those four bits with the control information to form a 16-bit control word. At every clock cycle of the 400 MHz clock, the outgoing SPI unpacks the 64-bit data into 16-bit data words and inserts control words by multiplexing onto 16-bit DDR bus at 400 MHz. If the re-assembly's control FIFO is empty, the outgoing SPI block sends idle control words onto SPI data bus. During regular interval, the outgoing SPI also sends training control word to provide skew detection.

Lookup Engine

This block receives ATM/Packet data and control from the SPI-4 receive interface block. It looks up a Flow ID and a set of parameters from a hash table in external SSRAM and forwards the data flow with the parameters to the segmentation engine.

Features

- Supports up to 1M flows.
- Supports up to 64 input ports.
- Operates at 200 MHz/5ns cycle time.
- Tables and configuration registers loaded via CPU Interface.
- Label extraction is programmable on a per port basis.
- Accepts packet SOP cells of 64 Bytes only.
- Accepts ATM cells of 52 Bytes only.
- Direct header stripping is programmable on a per port basis.
- Hash generator creates 4 hash values or slots in parallel from a key using 4 separate CRC polynomials as hash functions.
- Two bins per bucket or record to minimize collisions.
- There are two record memories for up to 8 read operations per a tag lookup.
- Hash table lookup constrained to resolve within 8 read operations.
- Overflow CAM of 32 X 72 entries for lookups that fail to resolve in 8 read operations.
- Fully pipelined lookup operations.
- Supports a maximum of two 8 X 72 M SSRAM. The normal size is two 2 X 72 M SSRAM.
- Supports the following flows with the following headers:
    - MPLS with PPP encapsulation.
    - LC-ATM that is VPI/VCI.
    - Switch headers.
    - FIDS.
- Operates in ingress and egress modes.

Functional Description

See Figure 83.

The lookup engine does label extraction, label hashing and table lookup for each data transfer that is SOP with a header. The type of traffic is unique to the incoming port. There will be no more than one type traffic on a port. The default type will get a special FID. The default FID is stored in a memory and it is specific to the input port. The lookup engine can strip the switch header of incoming cells.

Block Descriptions

See Figure 84.

CPU Hash Generation

The CPU through the interface can request a hash generation and get a hash index as a response. It will be used to set or tear down connections.

Port Configuration

The memory has two uses. It specifies the type of traffic on that port. Also, it has the Traffic parameters. The CPU either writes the parameters or the logic after lookup. If the CPU writes the parameters, then they are not changed until the traffic type is. They will be used in default traffic. If the traffic is direct, then the rest of the parameters in the memory for that port are not used. If the traffic is ENCAP type, then another set of registers are used. When the port traffic is none of the above, then the parameters values such as the type and FID are looked up from the record memory for SOP traffic. Then at the last stage they are driven on the interface bus. At the same time they are written immediately in the port configuration memory. For none SOP cells, the parameters are read from the port memory at the last stage and then driven on the bus. This will avoid data coherency problem in case the cells for the same port are contiguous. For ATM cells the SOP and EOP cell is the same.

DATA Coherency

The parameters on none SOP cells are read near the last stage of the data in order to avoid driving stale values on the interface bus. The CPU accesses to modify the memory are not tracked. The parameters used are the ones that are lookup or stored in the port memory.

If the SOP cell has a match on the last lookup, the parameters are written in the port memory near the end of the lookup cycle. If the next data cell is for the same port, then the parameters are not written yet in the corresponding memory location. Therefore, if the cell is not SOP, the port memory is read near the end of the pipeline of the cell lookup. See Figure 85.

CPU Accesses

The CPU accesses are allowed between lookup sequences when the memory bus is not busy.

External SSRAM

The Flow ID tables are implemented in off chip rams using 200MHZ, ZBT SSRAMS. These rams provide single cycle access with no 'dead cycle' penalty when mixing reads and writes. The table will contain a maximum of 8M entries. There is a

programmable register that will determine the size of the memory. The hash result will be trimmed to a number of bits that is programmable. There are two SSRAM channels provided. The usual size is 2M entries per channel. The memory can be accessed directly without hash for a read and write operations.

SRAM INITIALIZATION

The internal as well as the external ram will be initialized to

HEADER Selection

The header depending on the type has different sources. It can be transmitted with the traffic as in the case of direct or the special type. It can be found in the port memory as in the case of the default. Last, it can be found in record memory. See Figure 86.

Hash FIFO and Record Reading

See Figure 87. The data is stored until the port table is accessed. The label is generated. The data will continue to be clocked in a FIFO until the key is hashed and the index is calculated.

Memories

Input Port Parameters Memory

Each port will have a parameter entry in that table. The "type" parameter will determine if the rest of the entries are used. If the type is "000", then the rest of parameters of that memory entry are not used except for the header strip and tag extraction. The "HD STRP" indicates the number of bytes to strip from the header. The "INT HDR EXTRACT" indicates the number of bytes that represent the header. The header is at the end of the header stripped that is stripped. If the type is default, then the header to strip is zero. If the header to strip is not equal to zero, then the header will be stripped as in the case of the direct. The same argument is for the tag extract. The header will contain the FID, CLP, EFCI, and the OAM bits. The bits will be extracted and used in lieu of the look up data. The number of bytes of the header that are stripped and the parameter location is specified in the table. If the type is "001", then it is a default type and the rest of the parameters of that memory entry are used. If the type is "001", then the incoming traffic will be treated differently. The traffic is special and it is treated later in the specifications in detail. Again there are no hashing or lookup operations. If the type is different than the above three, then the tag is extracted, hashed and the parameters are looked up from memory. There are 64 entries in the table because the device can support a maximum of 64 ports. When there is a lookup operation, the parameters that are looked up are stored in the port memory and are used on the traffic until the next SOP cell. Infields where the hardware and the software perform the write operations, the software does it for the default traffic only, the other types of traffic are updated it by the hardware. See Figure 88.

The CPU writes the data in the port table entries for the default or the direct type. The logic will write the parameters for the port if it the type is not any of the above. The SOP cell will trigger the lookup. The rest of the cells on that port will use the table until the EOP or the next SOP. Then the parameters will change again.

In L2 encapsulation, if the CRC bit is set, then assert the CRC signal to the segmentation engine. See Figures 89 and 90.

Table Lookup Memory

The index or hashed key provided by the hash generator is used as an address. The number of bits that are used from the hashed key or index is programmable and depends on the size of the SRAM. The address is a base one and it is used to access four entries in the hash table located in external SSRAM. When these locations are read, the hash key is compared to the content of the memory. If the first location does not match, then there is collision and the second one is compared the same way. The process is repeated for all four entries. If there is no match and collisions resulted from the compare operations, then the process is repeated for the second hash result or index and so forth. If at the end there are only collisions, then the CAM is accessed and the process of comparison is repeated. If there is a hit, then in memory location there is Flow ID (FID), the QOS, and the rest of the control bits. Each hash table entry requires 72 bits. The bit definitions are summarized in Figure 91. The output port number is used to access a register. If the register bit corresponding to that port is set, then the EOP cell will have the error or discard flag set.

Hash Data Memory

See Figure 92. The data FIFO is a 15 X 104 bit wide FIFO. The FIFO compensates for the pipeline delay of the lookup path. It also provides buffering that is needed when the segmentation engine is not ready to accept any data. The data receiver block writes the FIFO. The data transmitter block reads the FIFO. The segmentation engine will assert a ready signal when it can accept data. When the ready is asserted, then the data is pushed to the segmentation engine.

Interface

See Figures 93 and 94. When the segmentation engine asserts the "ALMOST_FULL_SIGNL", the lookup engine will finish transferring the cell of eight transfers. Then at the cell boundary, the lookup engine will terminate transmission. At the same time the lookup engine will signal the SPI interface to terminate pushing data to the look up engine. The pipe of tag lookup operations is not violated. There are no lookup operations when the traffic flow is suspended.

CPU Interface

There are signals that are generated by the CPU interface and are used to control the enable and disable signals. See Figures 95 and 96.

Software Interface

See Figures 97 and 98.
The default value for the memory size is "001" which corresponds to 2 MEG.

FID Memories Select

See Figures 99 and 100. The default value for chip select is "00".

Register Description

The CPU interface doesn't support burst capability, instead, any read or write command will terminate with one address location; data bus is still 32 bits. Lookup Engine block has a fixed size of addresses allocated to it and all accesses to any internal/external memories or register is done via that specific address space. This method require the CPU Interface manager to have a mapping of the address spaces per block so that an incoming command will be redirect to the correct block.

Features

- Data bus of 32 bits.
- Single address access (not a burst access)
- All external memories and most of the internal memories are accessed via indirect accesses through registers
- Each block has dedicated addresses for its internal registers of up to 64 registers.

A command drives address and command (Rd or Wr) to the address FIFO and data if it's a write command. Each block starts it's own CPU access by POPING the address/command from the FIFO. It means that any CPU transaction starts when the address/command FIFO is not empty.

Address space

The address space is 64 entries only. The internal registers directly mapped to these addresses and external/internal memories are indirectly mapped. The address spaces per lookup engine are shown in Figure 101.

Direct/Indirect Accesses

A direct access is used to access. If there are internal memories or registers, which are longer then 32 bits, then read or write to these locations will be executed in more then one access. For example an entry in internal memory of 50 bits width (or a register with the same length) will be accessed in two CPU commands first access bits [31:0] and then [49:32]. Each register or small internal memories have their own address, which is unique to the entire chip set.

An indirect access is used to access internal and external memories. Each block has a set of registers dedicated for the indirect access. One register contains the address and the command (Rd, Wr, Init etc). Few other registers contain the data for a write command. Writing to the COM/ADD register triggers the execution of the command, once the block completed the execution of the command, the COM portion of the command is cleared by the block. When a read command occurs the data is written to the registers and then the CPU reads the registers to get the data.

Registers

See Figure 102.

Commands

NOP

See Figure 103.

Read LUT Memory1 (72 bits)

See Figure 104.

Write LUT Memory1 (72 bits)

See Figure 105.

Read LUT Memory2 (72 bits)

See Figure 106.

Write LUT Memory2 (72 bits)

See Figure 107.

Read Port Memory (44 bits)

See Figure 108.

Write Port Memory (44 bits)

See Figure 109.

Setup connection command for KEY

See Figure 110. This command sets up a connection by accessing external memories and initializes the corresponding locations to the proper parameters. It is a write command.

Tear-Down connection command for KEY

See Figure 111. This command tears down a connection by accessing external memories. It is a write command

GET HASH FID

See Figure 112. The response will be in the register. See Figure 113. This command tears down a connection by accessing external memories. It is a write command.

Read CAM Memory1 (72 bits)

See Figure 114.

Write CAM Memory1 (72 bits)

See Figure 115.

Read CAM Memory2 (72 bits)

See Figure 116.

Write CAM Memory2 (72 bits)

See Figure 117.

INIT ALL MEMORIES

See Figure 118.

INIT MEMORY 1 & CAM 1

See Figure 119.

INIT MEMORY 2 & CAM 2

See Figure120.

General commands

The following commands are combinations of the above simple commands. A user command like this should be broken to several Read or Write commands in the driver.

Initialized External memories

- Write "0" in all Record Memory 1.
- Write "0" in all Record Memory 2.

Initialized Internal memories

- Write "0" in all Port memory.

Principle of Operation

The lookup engine has to perform a port lookup before the FID lookup operation is performed. The memory is initialized and the valid bit is set for that FID. IF the valid bit is not set, then there will be no match. The lookup engine has the port of incoming traffic. As a result the following is done:

1. Read the port parameter entry by accessing the port parameter table.
2. Get the LOOKUP TYPE.
3. Extract the tag.
4. Perform the stripping, hashing, or lookup operations according to the type.

Once the parameter entry is read, then the lookup type is known. If the lookup type is *direct*, then the lookup engine will extract the FID, QOS, CLP, EFCI, and OAM if any and push all the control and data to the segmentation engine. If the traffic is *default*, then the parameters from the port table are used. If the traffic is *L2*, then the operation is handled as described in another section. If the traffic type is none of the previous ones, then the tag is extracted according to the type of traffic and then it is hashed. The result of hashing will be used to access the memory to get the rest of the parameters if there is a match. The memory content of the tag is compared to the hash tag. If they are equivalent, then the rest of the memory content is used for that FID, QOS, CLP, EFCI and OAM. If the tags do not compare then there is collision and the next hash result is used. There are four hash generators that will generate 4 indices. Two indices will be used for one memory and the other two for the second memory. Each record holds two bins. The total is 8 bins that might collide. If there is no match but collisions, then the overflow CAM is accessed. If there is collision or no record, then the packet or the cell is marked for CPU and a register is used for the FID.

CPU FIDS

If the keys stored in the memory locations do not match the hashed key, then the CAM content will be compared and the if it fails to compare, then the traffic will be sent to the CPU. There is a register that holds the FID and the rest of the bits necessary for the data traffic.

TAG Extraction

The lookup engine requires the minimum burst size of 16 bytes for the key extraction. The hardware extracts a contiguous number of bytes starting at an offset according to the traffic type of that port. The type of label to be extracted is determined by the port number. The location of the label is dependent of the type of traffic and is hard coded in the hardware.

ATM TYPES

If the type of the traffic is ATM, then the CLP, EFCI, and OAM bits are extracted and driven to the rest of the device. See Figure 121. Extract the above bits from header and drive to segmentation.

TAGS

The following Figures show the data formats for each of the supported protocols. The specifications are not hard coded. The port table will be used for the header's location. Each field is 8 bits wide with 12 bits VPI and 16 bits VCI field, for a total of 28 bits lookup tag. This 28 bits lookup tag is located in the 1st 64-bit word, bits 32 thru 63. See Figures 122 and 123. The PPP protocol is part of the link layer header. The tag is bit 60:41 of the first 64 bits word.

Port Types

The first lookup is to read the port parameters. If the type is "000", then the type is direct flow. There are no hash or lookup operations. The FID will be extracted from the incoming traffic itself. If the type is "001", then it is a default traffic. The default parameters are stored in the table itself. There are no hashing operations or lookup in this case. The third special case is the L2 type. The incoming header will be used to find the encapsulation header. There will be no hash and lookup operations either. The rest of the lookup types require a hash and lookup operation. The port table will specify the type of lookup that the lookup engine needs to perform on the incoming data. The lookup operation will assign an FID to the data stream or packet or flow, except in the bypass type. The FID is used only internally. See Figure 124.

Direct Header Format

See Figures 125-7. The SOP and EOP have to be extracted from the header. The SPI interface will also assert the SOP and EOP signals. The lookup engine will determine the set that uses and asserts to the segmentation engine from the type of the packet. In egress mode, the device uses the header set of SOP and EOP signals depending on the type. For example, if the switch fabric is a packet switch fabric, then the lookup engine will use the SPI interface set. If the switch fabric is a cell one, then the lookup engine will use the set parsed out of the header. When the header is stripped, then the number of valid bytes has to be adjusted accordingly. The data has to be packed in 64 bit transfers to the segmentation engine.

Special Header Addition

If the type is "010", then the flow will be handled differently. The incoming frame structure will look like the following:

See Figure 128. The CID and the CLASS will become the new FID of the flow. The rest of the incoming header will be used as the internal header. If the "SPCL LK" is

set, then the encapsulation header is added. If it is not set, then the header is extracted and no special header is added. The memory will have the structure depicted in Figure 129.

The memory location will be read and the length will indicate the number of valid bytes that have to be appended before the data as a special header. The header can be up to 16 Bytes long. The rest of the bits such as the EFCI, CLP and the OAM are driven, too as out of band signals. See Figure 130. The byte valid from the SPI interface to the segmentation engine has to be adjusted for the fact that the extra bytes have been added. The data has to be packed if the number of byte is not on 64 bit boundary.

Hash Index Generation

The extracted label is concatenated with the input port to produce the key. Keys may be padded out to 32 bits if they are less than 32. The key is hashed in hardware using the irreducible ETHERNET CRC32 polynomial listed below and three irreducible trinomial polynomials:

1. $G(X) = X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X^1 + 1$;
2. $G(X) = X^{31} + X^7 + 1$;
3. $G(X) = X^{31} + X^6 + 1$;
4. $G(X) = X^{31} + X^3 + 1$;

The key is hashed 32 bits at a time, requiring 1 cycle for all keys. See Figure 131. The type will determine if there is a need for hashing. The tag will be extracted in the first clock because it is available in the first 64 bits transfer. The key will be formed. The key will then be hashed using four different hashing functions. The hashing is done in parallel. The first hashed key or index will be used as a base register to read two memory locations. As soon as the first location is read, then the key that is stored in that memory location is compared to the hashing key. If there is a match, then the read operations are halted. The content of the memory of the matching key will be used. If there is no match, then the next location content will be compared and so forth. The second index will be used to read the third and forth records if the first record of two bins fails to get a match. Then the third and the forth indices are used for the second memory. If all four indices fail to get a match, then the CAM is accessed. If the CAM contents do not match, then the traffic is forwarded to the CPU.

Timing Diagrams

See Figure 132. The transfer size is assumed to be 64 bytes for up to eight transfers for SOP SOB data traffic.

Segmentation

The segmentation engine segments the incoming packets into 52-byte ATM, 48 bytes AAL-5, or switch cells and stores them in memory as cells of 64 bytes. The segmentation engine adds the necessary number of pad bytes to complete the cell.

Main Features

- Segments incoming data from SPI-4 on per port basis, supports maximum of 64 ports
- Segments incoming data from CPU interface

- 64K bytes maximum packet
- Segments, maps or de-maps incoming data into either ATM/AAL-5 cells or switch cells
- Stores segmented cells in memory as cells of 64 bytes
- Supports the traffic depicted in Figure 133.
- Types of segmentation and applications for various traffic types.
- Backpressures the scheduler in the egress side from the per flow Q in the ingress side via status cells through a serial interface to segmentation engine.
- Calculates CRC-32 for packets segmented into AAL-5 cells.
- Calculates new CRC-32 for L2 header.
- Supports programmable maximum transfer unit up to 16K bytes for packet data.
- Each port has a 32-bit counter to count packets coming from that port.
- Each port has a 48-bit counter for data through that port.
- Tags bad packets that they can be discarded later.
- Internal memories to store parameters and data.
    - Dual-port SRAM
        - Control - 65 x 39 bits to store packet length, and cell byte count
        - Data - 640 x 64 bits to store data
        - CRC - 65 x 32 bits to store the partial CRC for each port
        - Statistic - 65 x 80 bits to store packets and data received for 64 ports, and one for CPU
    - Single-port SRAM
        - Buffer SRAM - 80 x 7 bits to maintain the free buffer list for data
        - Queue - 80 x 70 bits to queue information for the complete cells
    - Operates at 200 MHz

Functional Description

See Figure 134. There are three input data paths to the segmentation engine: data from SPI-4, CPU, or serial interface. There are two output data paths from the segmentation engine: to memory manager, and serial interface to the scheduler.

See Figure 135. The segmentation engine receives data from the incoming SPI-4 during normal operation. Based on the type of application, the segmentation engine segments, de-maps, or maps the incoming data to either ATM/AAL-5 or switch cells. As shown in Figure 133 there could be five different types of segmentation (as opposed to types of application) performed by the segmentation engine. The following explains the operation of the segmentation engine for each of these types.

TYPE 1 Segmentation

See Figure 136. Type 1 segmentation is for application type 0, 4, 8, 9, 10, 11,12, and 14. The segmentation engine receives the 64-byte cells from the look up engine and queues the cells to the output control block when the entire cell has been received.

TYPE 2 Segmentation

See Figure 137. Type 2 segmentation is for application type 1 and 5. The segmentation engine receives the 64-byte cells from the look up engine and queues the cells to the output control block when the entire cell has been received. The cells come in as 52 bytes ATM data with 12 bytes of pad and output as 48 bytes of AAL5 data with 16 bytes of pad to the memory manager block. The segmentation engine removes the 4 bytes ATM header and appends 4 bytes of pad to the end of the cell.

TYPE 3 Segmentation

See Figure 138. Type 3 segmentation is for application type 2 and 13. The segmentation engine receives the 64-byte cells from the look up engine and queues the cells to the output control block when the entire cell has been received. The cells come in as 64 bytes packet data and output as 48 bytes of AAL5 data with 16 bytes of pad to the memory manager block. The segmentation engine segments the incoming data into 48 bytes, appends 16 bytes of pad, and passes the control information for the completed cell to the output control block. CRC-32 is calculated when the cell is being sent out to the memory manager block.

TYPE 4 Segmentation

See Figure 139. Type 4 segmentation is for application type 3 and 6. The segmentation engine receives the 64-byte cells from the look up engine and queues the cells to the output control block when the entire cell has been received. The cells come in as 64 bytes packet data and output as 64 bytes of AAL5 like data to the memory manager block. The segmentation engine segments the incoming data into 64 bytes and passes the control information for the completed cell to the output control block. The segmentation engine calculates CRC-32 and appends an AAL5 like trailer at the end. However, CRC-32 is calculated when the cell is being sent out to the memory manager block.

TYPE 5 Segmentation

The segmentation engine actually is not segmenting data in type 5. When the ingress segmentation engine receives a status cell from the egress PFQ through the serial interface, it queues that status cell to the memory manager like any other cells. The status cell is sent through the switch to the egress segmentation engine, and the egress segmentation engine sends the status cell to the ingress scheduler through a serial interface. See Figure 140. The serial input and output data paths provide a way for the per flow Q on the egress side to back pressure the scheduler on the ingress side from sending data.

CPU injected packets

The CPU can inject data through software command as described later in section 1.6.2. When the segmentation engine detects the CPU inject packet command it sends the cell out in the next control pop from the memory manager. However, there can only be no more than one CPU cell sent within 64 cells time. The segmentation engine starts a cell counter whenever it sends a CPU cell, and no more CPU cells will be queued out until the counter reaches 64.

As shown in Figure 141, the CPU can write and read the internal registers inside the segmentation engine through the CPU interface. The CPU can also input packets into the segmentation engine through the CPU interface.

Block Description

See Figure 142.

Description of INPUT CONTROL block

In normal operation, when the segmentation engine has room in its FIFO for incoming cells it asserts a FIFO not full signal to the look up engine and waits for a valid signal from the look up engine to indicate there is valid control information on the bus. When there are valid controls, the input control block performs read from various internal memories:

a) Reads from the FBL SRAM to get two pointers to the data SRAM,
b) Uses the port address to read the SEG_TBL SRAM to get the per port parameters.
c) Uses the port address to read the STATISTIC SRAM to get the statistics for that port,

The per port parameters (SOP, EOP, etc.) from the bus are compared with those from the SEG_TBL, and error conditions are set accordingly. The port address from the bus is compared to the CUR_PID, which stores the last port address processed, to see if the incoming burst is from the same port address. If so, it means the data burst is the continuation to the one previously processed, and the control information in the internal registers are used instead of those read from the memories. Otherwise, it means the incoming data burst is for a new port address and new controls are to be used, and the controls for the last processed one are to be stored before the new ones are loaded. Detailed information is discussed later in the timing flow.

The input control block has two sets of control parameters, one for the last processed port address, i.e. the current parameters, and one for the port address from the look up engine that comes with the valid signal.

The controls, statistics, and the two pointers are then passed to the data block with a start signal to start the data state machine.

The input control block also receives FIFO status from the FBL and QUEUE blocks and generates the FIFO not empty signal to the look up engine.

Description of FBL block

The FBL (Free Buffer List) block keeps track of the open entries in the data SRAM. After power on reset, the CPU needs to write the init bit in the segmentation register to start the self- initialization of the free buffer list. The self-initialization takes a little over 80 cycles, and no data is allowed through the segmentation engine block until the initialization is done.

The FBL block receives the read strobe from the control block. For each read strobes the FBL block reads two locations from the FBL SRAM and passes the two pointers back to the control block.

The FBL block receives the write strobe and the return pointer from the data block. It writes the return pointer to the write address when there is no read pending.

The FBL block generates a full status to the control block if there are no more free entries in the data SRAM.

Description of FBL SRAM block

This is an 80 x 7 bits single-port internal SRAM to store pointers to the free memory blocks in the data SRAM. The CPU needs to set the init bit to start the self-initialization after reset.

Description of SEG_TBL SRAM block

This is a 65 x 19 bits single-port internal SRAM to store the parameters for the 64 ports plus CPU. This RAM is accessed twice by the control block for each cell comes in for a particular port, one read to retrieve port parameters and one write to return parameters for that port.

Description of STATISTIC SRAM block

This is a 65 x 80 bits dual-port internal SRAM to store number of packets (32 bits) and data bytes (48 bits) received on the 64 ports and CPU interface for statistic purpose.

Description of DATA PIPELINE block

The DATA PIPELINE block delays the incoming data and byte valid to match the delay for control signals through the INPUT CONTROL block.

Description of DATA block

The data block receives the control information from the input control block and decodes the types of application to be performed by its input state machine. The input state machine starts when the data block receives the start signal from the control block. The input state machine has five different branches based on the segmentation types decoded from the application types. For each of the branch the input state machine performs the following operations and writes the data into the data SRAM:

a) Add 8 bytes of pad
b) Add 16 bytes of pad
c) Segment incoming data into 48 bytes and add 16 bytes of pad
d) Segment incoming data into 64 bytes
e) Pass through the 64 bytes

The main function of this input state machine is to accumulate a 64 bytes cell for each port address. When there is a completed cell, the input state machine generates a control signal to queue the control information for that particular cell into the QUE FIFO in the queue block. At the same time, the data block writes the statistics back into the STAT SRAM and sends the parameters back to the input control block.

There is a data byte counter which increments for each incoming data bytes to count for the 64 bytes cell. For a cell with EOP in it, the counter value is sent to the memory manager indicating the number of bytes valid in the EOP cell.

For AAL5 or like data cell, the data block also check to see if a new cell is needed for the trailer. If needed so, the data block writes the data/trailer to the buffer entry indexed by the second pointer. Otherwise, the data block returns the second pointer to the end of the free buffer list.

There is also an output state machine, which starts when the queue block requests to de-queue from the data block. This output state machine receives the pointer to the data

SRAM and the control information to be processed on the cell. The output state machine reads from the data SRAM with the buffer pointer and output the data to the memory manager. When the data block finish sending data to the memory manager it returns the pointer back to the free buffer list.

If the control information indicates a need to generate CRC-32, the data block reads the CRC SRAM with the port address from the control information and load the partial CRC into the CRC-32 engine. At the end of the cell and not EOP, the data block writes back the partial CRC into the CRC SRAM. If the control indicates a need to calculate new CRC for the L2 header, the output state machine replaces the existing CRC bytes from the packet with the ones calculated with the new L2 header, and the new values are sent through the CRC engine to calculate for the AAL5 trailer.

There are two statistic counters in the data block, one for the number of packet received and the other for the number of data received. These statistics are kept on a per port basis, where the data block load the information from the STAT SRAM with the load signal from the input control block.

Description of DATA SRAM block

This is a 640 x 64 bits dual-port internal SRAM to store the incoming data as the segmentation engine completes segmenting them into 64-byte cells. The memory is allocated into 80 blocks with each block has eight entries, total of 64 bytes per block.

Description of CRC_GEN block

This block generates the CRC that covers the PDU (Protocol Data Unit) in the AAL-5 format. The polynomial used is CRC-32 as follows.

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

Since the internal data bus has 64 bits, the CRC-32 is calculated in two portions:

a) CRC-32 engine with 64-bit data input

b) CRC-32 engine with 32-bit data input

From the beginning of the data, CRC-32 engine with 64-bit input data bus is used all the way to the end of the data, i.e. right before the trailer for AAL5. Then it loads the remainder from the CRC-32 engine with 64-bit input data bus to the CRC-32 engine with 32-bit input data bus and calculates the CRC with the four MSB bytes of AAL5 trailer. The CRC_GEN block reads the partial CRC with port ID as the address to the CRC SRAM. The CRC_GEN block loads the partial CRC into the CRC engine and starts calculating as the data are going out to the memory manager. When the cell is sent to the memory manager block, the CRC_GEN block writes back the partial CRC into the CRC SRAM, again using port ID as the address. At the end of a packet, the CRC is muxed out as part of the AAL-5 trailer. For L2 header, CRC is calculated for the packet and is used to replace the CRC bytes in the packet before calculating the CRC for the AAL5 trailer.

Description of CRC SRAM block

This is a 65 x 32 bits dual-port internal SRAM to store partial CRC for each port. The data block reads and loads the partial CRC if the traffic type requires CRC-32 calculation. The data block writes back the partial CRC when the completed cell is sent to the memory manager.

Description of OUTPUT CONTROL block

The OUTPUT CONTROL block sends the available signal to the memory manager to indicate that there are completed cells available to transfer. The memory manager replies with the control pop signal if it is ready to receive the completed cells. When there is control pop from the memory manager, the output control block reads the QUEUE FIFO for the control information for that cell and passes them to the data block. The data block performs read to the DATA memory as described in the data block above. The output control block sends the control information lining up with the output data. The output control has to work in compliance with the global sync, it only asserts the available signal in second clock slot from the global sync (see detailed timing in later section). The memory manager is to responds in the third clock slot, and the output control block generates the pop signal to the QUEUE FIFO in the following clock. The output control block also generates the read signal to the data memory in the same clock slot.

If there is status cell pending, the output control block sends the status cell using one of the regular port accesses from the global sync, and the skipped cell takes over the CPU access during that particular global sync. This means CPU access is skipped for the global sync that has a status cell to send. The output control block returns to its normal port access in the following global sync.

Description of Q_FIFO SRAM block

This is an 80 x 70 bits single-port internal SRAM to store control information for cells queued to the memory manager block.

Description of CPU_INTF block

The CPU_INTF block is to interface with the external CPU. It supports CPU direct access to the internal registers and indirect access to the internal memories. CPU can also injected packet data through this interface. The CPU can inject packet/data through use of CPU inject data command, however, it can only injects a 64 bytes cell one at a time. As soon as there is a complete cell, the CPU_INTF block asserts a signal to the output control block to output the cell in the CPU access.

Description of PFQ_SEG INTF block

The PFQ_SEG INTF block receives data from the serial interface between per flow Q and the segmentation engine. The PFQ_SEG INTF block shifts the serial data into a 64-bit shift register and load the output of the shift register into an internal register. The control information is extracted from the serial data. When the PFQ_SEG INTF block receives the entire status cell, it queues the completed cell into the queue FIFO.

Description of SEG_SCH INTF block

The SEG_SCH INTF block receives data from the data block and outputs the data through a serial interface to the scheduler block. When the segmentation engine receives a status cell from the incoming SPI-4 interface, the data block outputs the data into the SEG_SCH INTF block instead of the data SRAM. As soon as the complete cell is

received, the SEG_SCH INTF block shifts the cell out the serial interface to the scheduler.

ERROR CONDITIONS

Several errors can be detected in the segmentation engine, and the error packet will be marked error by the SEG_ENGINE. The different errors are in Figure 143. These errors are reported in the status register. There are individual bits in the register to mask the individual errors from generating interrupt to the CPU.

Memory Description

Summary of Internal Memories:

See Figure 144.

Data structure for SEG_TABLE

See Figure 145. The SEG_TABLE is a 65 x 39 bits internal SRAM addressed by the port ID (PID). The segmentation table has 65 entries, one for each port and CPU.

BAD : This bit is set if detects error condition and needs to discard
SOP : This bit is set if start of packet is received for the port
EOP : This bit is set if end of packet is received for the port
Cell Byte Count : This field keeps track of the byte count received for the 64-byte cell
PTR : The start address to the cell in the data SRAM
Packet Length : This field keeps track of the total length for the incoming packet for the port
CLP : Cell Loss Priority, valid only with ATM traffic
EFCI : EFCI, valid only with ATM traffic
CALC_CRC : Calculate L2 CRC.
OAM : indicate OAM cell.

Data structure for FREE_BUFFER

See Figure 146. The FREE_BUFFER is an 80 x 7 bits internal SRAM that used to store the pointer to the free data block in the data SRAM. The segmentation engine takes a pointer from the top of this table to receive the incoming data. When the memory manager gets a complete cell off the queue the pointer to that cell is return to the end of this table.

Data structure for Q_FIFO

See Figure 147. This 80 x 70 bits internal SRAM that stores the queue of complete cells and the information for each queued cell.

FID : Flow ID of the incoming data for the port.
TYPE : Traffic type of the incoming data for the port.
BAD : Mark the packet to be discarded.
SOP : Mark the start of the packet.
EOP : Mark the end of the packet.
CELL_LEN : Indicate the number of valid bytes in the 64-byte cell.
PTR : The start address to the cell in the data SRAM.

PKT_LEN : Length of the packet, valid only for packet traffic.
QOS : Quality of Service.
O_PORT : Output port number.
CLP : Cell Loss Priority, valid only with ATM traffic.
EFCI : EFCI, valid only with ATM traffic.
CALC_CRC : Calculate L2 CRC.
OAM : indicate OAM cell.

Data structure for DATA SRAM

See Figure 148. The DATA SRAM is to store the incoming data in 64-bytes block.

Data structure for CRC_TABLE

See Figure 149. The CRC Table is a 65 x 32 bits dual-port internal SRAM addressed by the port ID (PID). Each port has an entry to this SRAM to store the partial CRC generated for the AAL-5 traffic.

Data structure for STATISTIC

See Figure 150. The Statistic SRAM is a 65 x 80 bits internal SRAM addressed by the port ID (PID). Each port has an entry to this SRAM to store the number of packets and data received for that port. Statistics for the CPU interface is in the last entry. The CPU interface accesses the content of this SRAM.

PKT_RCVD : Number of packets received for each port
DATA_RCVD : Number of data bytes received for each port

Interface

Interface Ports

System

See Figure 151.

Segmentation ⇔ Lookup

See Figure 152.

Segmentation ⇔ Memory Manager

See Figure 153.

Segmentation ⇔ PFQ

See Figure 154.

Segmentation ⇔ SCH

See Figure 155.

Segmentation ⇔ CPU Interface

See Figure 156.

Timing Diagrams

See Figure 157.

Memory Access

See Figure 158. The bolded "**Rd**" indicates that there are data dependencies for these reads. The current PID being processed is compared with the incoming one, the segmentation uses the current per port parameters if they are equal. Otherwise, it uses those read from the memories.

Registers Description

Registers

See Figure 159.

Commands

Summary of commands

See Figure 160.

CPU Inject Packet

See Figure 161.

Read SEG FBL memory

See Figure 162.

Write SEG FBL memory

See Figure 163.

Read SEG CRC memory

See Figure 164.

Write SEG CRC memory

See Figure 165.

Read SEG STATISTIC memory

See Figure 166.

Write SEG STATISTIC memory

See Figure167.

Read SEG QUEUE memory

See Figure 168.

Write SEG QUEUE memory

See Figure 169.

Read SEG DATA memory

See Figure 170.

Write SEG DATA memory

See Figure 171.

Read SEG TBL memory

See Figure 172.

Write SEG TBL memory

See Figure 173.

Principles of Operation

The segmentation engine has three different sources for its incoming data: SPI-4, CPU interface, and status cells from per flow Q. The control information for the complete cells are queue into the queue FIFO, and are output to the memory manager following global sync scheme. If there is a status cell from per flow Q, it will be output in the next available cell slot, and the CPU slot will be used to compensate for the port that its slot is taken by the status cell. The segmentation engine resumes to the global sync in the following global sync cycle.

In normal operation, the segmentation gets its data from the SPI-4. If the FIFO full signal is not asserted, it means the segmentation engine is ready and has room in its FIFO to receive data from the SPI-4. The look up engine then sends the control information together with the data to the segmentation engine. For the first data cell of a packet for a particular port the segmentation engine sets the SOP bit for that port in the segmentation table, writes data into the data FIFO, increments both the packet length and cell count as data come in. The data state machine keeps receiving data from the look up engine until the end of data burst, complete cell size, end of packet, or if there is data coming from the CPU or serial interface.

If it is the end of data burst only, the segmentation engine loads the packet length and cell count back into the segmentation table for that port. The segmentation engine is now ready to receive the next data burst.

If the segmentation engine has a complete cell size, it queues the cell by writing the cell information into the Q_FIFO. The segmentation engine resets the cell count, loads the packet length back into the segmentation table. Again, the segmentation engine is now ready to receive the next data burst.

If it is the end of a packet, the segmentation engine checks to see if the traffic type is for packet to AAL-5. If so, the segmentation engine checks to see if another new cell is needed for the AAL-5 trailer. The segmentation engine then queues the completed cell(s) into the Q_FIFO, and then resets the entire entry for that port in the segmentation table. For packet data to AAL-5 cells, the CRC_GEN calculates the ATM CRC-32 when the cells are going out to the memory manager. The partial CRC is stored on per port basis, and the partial CRC is loaded back into the CRC generation engine for each cell queued out the particular port.

The segmentation engine can process multiple open ports. For each port ID it reads from the segmentation table the parameters for that port, receives data for that port, stores the parameters back into the table, read the parameters for a different port from the table and so on.

The segmentation engine queues a completed cell into the Q_FIFO and asserts the SEG_MEM_AVAILABLE signal to the memory manager block. The memory manager block pops the Q_FIFO to get the information (FID, PID, etc.), and the segmentation engine reads the data from the data SRAM using the pointer from the Q_FIFO. The segmentation engine sends the data to the memory manager with timing according to the global sync. After the segmentation engine sends the completed cell to the memory manager, it returns the pointer to the FBL.

If the incoming data is from the CPU interface, the segmentation engine receives data from the CPU_FIFO, strips the header from the cell and extracts the control

information from the header. The segmentation engine then waits to complete the cell and queues the completed cell to the Q_FIFO as in normal operation.

If the incoming data is from the serial interface from per flow Q, the segmentation shifts the serial data in, packs to 64-bit bus and store in an internal FIFO. The header of the incoming cell is stripped and control information such as flow ID and traffic type are extracted from the header. The segmentation engine waits to complete the 64-byte cell and queues it into the control FIFO (Q_FIFO). When the segmentation engine detects a status cell as the traffic type coming through the SPI interface it queues the complete cell into a separate FIFO in the SEG_SCH_INTF block and send the data out in a serial format to the scheduler.

Memory Manager

Features

The Memory Manager (MM) block manages all the data and control traffics from/to the Segmentation, Per Flow Queue, Reassembly, and external SDRAM (or DRAM). Input cells (data) are sent from the Segmentation block to the MM block. The ATM cells and packets are sent to the Memory Manager block in the "internal cell" format. Internal cell size is determined at the power CPU of the chip (i.e. programmable). Figure 175 depicts the top level interface of the Internal Memory Manager block with others.

The Memory Manager block has several main features listed as follows:

1. Support up to 512 Mbytes of external storage Memory using DDR-SDRAM.
2. Cache the incoming "bypass" cells so that they can be transferred to the Reassembly block with minimum latency (without going through the external SDRAM path).
3. Act as a buffer for data going from the Segmentation block to the external SDRAM.
4. Act as a buffer between the incoming burst from the external SDRAM and the output to the Reassembly block.
5. Receive Cell Data and Control Information from the Segmentation block and initiate En-queuing process.
6. Receive De-queuing information (such as BID, FID) from the Per Flow Queue, get requested data from either the Internal Memory or the external SDRAM, and then send them to the Reassembly block along with other related information.
7. Interface with the Free Buffer List (FBL) inside the Per Flow Queue block:
    a. Send En-queuing request to FBL to receive a BID indicating the En-queuing cell's data location in DDR-SDRAM.
    b. Send De-queuing BID to FBL to release the location pointed by the BID.
8. Manage the processes and orders of reading from the SDRAM and writing to the SDRAM to maximize the overall bandwidth of the SDRAM. This

relates to the strategy of selection of banks of memory to read and write to reduce the overhead penalty.

9. CPU Interface: All memories of the Memory Manager can be accessed (read/write) by the microprocessor through the CPU interface:
   a. Internal Memory
   b. CAM, Internal Memory List (IML), Free Internal Memory List (FIML)
   c. external SDRAM (by Queuing CPU read/write requests)
10. Frequencies: 200 Mhz

See Figure 174.

Functional Description

Refer to Figure 174 for the top level block diagram of the Memory Manager (MM) block and Figure 175 for the block diagram of the Internal Memory Manager (IMM). The incoming cells are sent to the Internal Memory Manager block by the Segmentation block along with other information for the En-queuing process. After getting the En-queuing BID from the Free Buffer List block, the IMM synchronizes the BID with all the information from the Segmentation block and then send them to the Per Flow Queue block for En-queuing.

Incoming cells are temporarily stored in the Internal Memory (internal SRAM) that is controlled by the Internal Write Manager block. If cells are specified as "bypass" indicated by the Type input, they stay in the Internal Memory and then they are sent to the Reassembly block directly during the De-queuing process. If incoming cells are specified as "non-bypass" indicated by the Type input, they are temporarily stored in the Internal Memory, and then moved to the external SDRAM. The moving of all "non-bypass" data is handled by processing all the SDRAM write requests with En-queuing BID/IMBID inside the SDRAM Write Request Queues block. During the De-queuing process, if some of these cells are called to be de-queued (i.e. CAM Hit) before they are sent to the SDRAM, their status will be changed in the CAM and they will stay in the Internal Memory instead of moving out to the SDRAM. These cells will be sent to the Reassembly block directly. Right after a cell is taken out of the Internal Memory, the CAM and the Free Internal Buffer List are Updated immediately.

During the De-queuing process, the Per Flow Queue interfaces with the Internal Memory Manager block with a De-queuing BID. The IMM will check in the CAM to see whether the requested cell (data) is inside the Internal Memory (SRAM) or inside the external SDRAM. If CAM indicates CAM Hit and the requested data is a "bypass" type, it means that the requested cell is inside the Internal Memory. In that case, the Memory ID Manager (MIM) will issue a command to the Internal Read Manager to read the CAM hit data out of the Internal Memory; the IMBID (internal buffer ID) will be provided to the Internal Read Manager. The MIM also modifies the CAM table so that this specific data will not go to the external SDRAM (it will be marked as "bypass") and it signals the De-Queue Request Manager block not to send a read SDRAM request to the SDRAM Read Request Queues block.

During the De-queuing process, if the De-queuing BID can not be found inside the CAM, it means that the requested data is inside the external SDRAM. In this case MIM will signal the De-Queue Request Manager block to send a read SDRAM request to

the SDRAM Read Request Queues block with the BID and IMBID (provided by the MIM block from the Free Internal Buffer List). The SDRAM Read Manager block takes read requests from the SDRAM Read Request Queues and processes them to through the SDRAM Manager (outside of the Internal Memory Manager block). Read-back data from the SDRAM will go through the SDRAM Read Manager block with the data itself, the BID and the IMBID assigned to it previously. The data and IMBID will go to the Internal Write Manager which will open the gate for the data to be stored inside the Internal Memory block at the IMBID location. As data enters into the Internal Memory, the CAM table will also be Updated properly (it will be marked as "bypass"). The Internal Read Manager then will schedule the data to go out to the Reassembly properly based on the order of the De-queue requests came in previously (stored in the De-queue Queue block).

Microprocessor can access (read/write) the external SDRAM through the CPU data bus for testing and diagnostic purposes. The access can be random and data are not managed or shaped. The CPU Interface block inside the Memory Manager sends read or write requests from/to the SDRAM Read Request Queues and the SDRAM Write Request Queues, respectively. The CPUID/BID will be queued along with the IMBID/BID. The SDRAM Read Manager and the SDRAM Write Manager will send these information and associated data to the SDRAM BW Optimizer based on the bank selection strategy inside it. Data coming back from the SDRAM is routed properly to the CPU Interface block based on the returned CPUBID/BID information.

Block Description

Internal Memory Manager

Refer to Figure 175 for the block diagram of the Internal Memory Manager.

Internal Memory Block

This is an internal dual-port SRAM memory. It can run at 200 Mhz with 5 ns access time. The input data bus width is 128 bits and the output data bus width is also 128 bits. The input data, IMBID and control signals are provided by the Internal Write Manager block. The output data go to the Internal Read Manager in response to the read request along with IMBID provided by the Internal Read Manager block.

The size of the SRAM is currently defined as 64 Kbytes (= 4K x 128 = 1K of 64-byte cells).

Memory ID Manager (MIM) Block

This block consists of a CAM, a Free Internal Memory List (FIML), an Internal Memory List (IML), and a MIM (Memory ID Manager) Controller. Refer to Figure 3 and Figure 4 to understand the mechanism of the CAM, FIML, and the IML blocks. FIML and IML are internal SRAMs. The CAM contains the BID values. The MIM Controller block manages all the incoming commands to Update the CAM, FIML, and IML blocks.

It also arbitrates all the Update commands from different blocks as they can happen at the same time.

Free Internal Memory List (FIML):

This list contains 1K entries (i.e. to support 1024 cells inside the Internal Memory (SSRAM)). The content of each entry is the IMBID that is an address points to the Internal Memory List (IML) to get IMBID and status. It also corresponds to the output address of the CAM. The FIML contents are initialized from 0 to 1023 when an initialization command is received from the CPU interface.

The FIML has 2 pointers: TakePtr and ReturnPtr. TakePtr and ReturnPtr are initialized at the top of the List, ReturnPtr points to location 0 and TakePtr points to location 1. TakePtr points to a free IMBID value. TakePtr is incremented by 1 every time an IMBID is taken. ReturnPtr points to a not-available IMBID value. Every time a location is returned to the FIML, a new (i.e. returned) IMBID value will be written into the location that ReturnPtr currently points to. Now that location represents a free cell with the returned IMBID points to a free cell inside the Internal Memory. ReturnPtr is also incremented by 1 every time an IMBID is returned.

When TakePtr and ReturnPtr reach above the value of 1023, they will be rolled over to the value of 00. TakePtr and ReturnPtr will never be equal. If (TakePtr – ReturnPtr) is equal to -1, then the IM is full and the IM_Full bit of the Mem_Status_0_reg will be set. If (TakePtr – ReturnPtr) is equal to 1, then the IM is empty and the IM_Empty bit of the Mem_Status_0_reg will be set. The status of Full-1 is also recorded in the IM_Full_1 bit.

The CPU interface should be able to write and read to/from the FIML memory:

1). CPU_Wr_FIML{input: Adr, Wr_value}:

This command allows the CPU interface to write to the FIML table where Adr is the address value (range from 0 to 1023) and Wr_value is the content that we are going to write to. Refer to Register Description section.

2). CPU_Rd_FIML{input: Adr; output: Rd_value}:

This command will read the content of the FIML table at location Adr. Refer to Register Description section.

Internal Memory List (IML):

The IMBID value is the address of the IML block. The content of each entry contains 3 bits: the valid bit, bypass bit, and readback bit.

- Valid: this bit indicates that at IMBID location in the Internal Memory, a cell is valid.
- Bypass: IMBID points to a cell that is whether a bypass or non-bypass type. A bypass type cell will not go into the external SDRAM, it is ready inside the IM to go out to the Reassembly. A non-bypass type cell is in transition to go into the external SDRAM.
- Readback: indicates that this cell is from the SDRAM and ready to go out to the Reassembly block.

CAM:

CAM contains the values of BIDs (external SDRAM pointers) that are inside the Internal Memory. If a CAM hit occurs, an IMBID value will be returned by CAM. After receiving the IMBID, the MIM will go to the IML and read the content at IMBID. This content contains IMBID (Internal Memory address) and its status. If valid bit is equal to 1, it means that IMBID is valid and, therefore, this CAM hit is valid. Otherwise, a "miss" is returned.

There are 4 main internal commands (between blocks and not CPU commands) can be issued to the MIM block (from other blocks):

1). Take_ID ← IMBID:

In this command, the MIM will access the FIML and take a free IMBID from it. The output of the command will be a free IMBID that represents a free cell inside the Internal Memory. The TakePtr will be incremented by 1 after it releases a free cell.

2). Return_ID{input: IMBID}:

In this command, the MIM will receive an IMBID as an input. It will return this pointer to the FIML. The ReturnPtr will be incremented by 1 after an IMBID is returned.

3). Compare_CAM{input: BID; output: IMBID, status, hit/miss}:

In this command, input argument is BID and output is IMBID, status (valid bit, bypass bit, busy bit) and hit/miss bit. The MIM receives BID and compares it to the entries in the table. If a match is found, the CAM will output the corresponding IMBID. The MIM goes to the IML and read its content at the location IMBID. The MIM also checks to see if the "valid" bit is equal to 1 at this location. If it is equal to 0 then a miss has occurred, otherwise, it is a CAM hit. The content of IML at IMBID will include status, and a hit/miss bit.

4). Wr_CAM{input: IMBID, BID, status}:

This command will modify the content of the CAM table and the content of the IML table with the new values of BID and status, respectively.

5). Rd_CAM{input: IMBID; output: BID, status}

This command reads the contents of the CAM table and the IML table.

The CPU interface should be able to write and read to/from the CAM table:

1). CPU_Wr_CAM{input: IMBID, BID, status}:

This command allows the CPU interface to write to the CAM table and the IML table with new values of BID, and status, respectively.

2). CPU_Rd_CAM{input: IMBID; output: BID, status}:

This command will read the content of CAM table (i.e. BID) and the content of IML table (i.e. status)

Internal Write Manager Block

This block receives input from the Segmentation block and the SDRAM Read Manager, and manages the writing process into the Internal Memory (SRAM) block.

There are 4 main functions that this block handles independently:

Function 1: (Segmentation→ Internal Memory).

1) Segmentation asserts Seg_Mem_Available signal to the MM block to let it know that there is data available in its fifo.
2) MM block sends Mem_Seg_Cntl_Pop signal to pop the cell control information of a cell.
3) Cell control information is sent to the MM block.
4) At a fixed delay from the time that IWM gets the cell control information from the Segmentation block, the Segmentation block starts to send cell data to the MM block at 128-bit word at 10ns period (every other cycle of 200 Mhz clock). The 128-bit data word will be first latched into flip-flops inside the IWM.
5) From FIML, get free IMBID: issue Take_ID command to MIM block.
6) Write data into the Internal Memory: Wr_IM{IMBID; Seg_Data}.
7) Update CAM with new BID: Wr_CAM{IMBID; BID, valid=1, bypass=0/1, readback=0}.
8) If (not "bypass"),
    a. Then put new BID/IMBID into end of SDRAM Write Request Queues: Wr_SWRQ{end of queue; BID/IMBID}.

Note: This block has a small FIFO to receive data inputs from the Segmentation block. The size of the FIFO is 4 64-byte cells with 128 bit wide. Data will be stored in this IWM FIFO before it is sent to the Internal Memory.

Function 2: (Segmentation➔ Per Flow Queue).
1) IWM sends the cell control information to the PFQ block with a command specified in Mem_PFQ_Com signals (i.e. "00": invalid; "01": enqueuing; "10": discard; "11":release (for dequeuing)).
2) PFQ receives the information, processes it and then
    a. If command = "01", then it sends back en-queue BID (Buffer ID) to the MM block 8 cycles later. This BID will be used as an address to write data into the external SDRAM. If BID = "7FFFFF", then that particular cell will be dropped and not written into the SDRAM.
    b. If command = "10", then it sends BID = "7FFFFF" back to the MM block. In this case, data will not be written into the external SDRAM.

Function 3: (Control data path to Internal Memory).
Inputs and outputs of the Internal Memory are time shared as follows (to handle dual-port 128-bit wide SRAM at 200 Mhz):
1) Slot 0 (period = 5ns): input from Segmentation.
2) Slot 1 (period = 5ns): input from SDRAM Read Manager.

Function 4: Microprocessor (CPU) Access.

When switched to this mode, the microprocessor can write to the Internal Memory (IM) with a specified address. During this mode, there is only the CPU interface block that can access (read/write) to the IM block. Refer to the Register Description section..

CPU_Wr_IM{input: Adr, Wr_value}: write to IM at the Adr location with the value of Wr_value.

SDRAM Read Manager Block

This block receives read requests from the SDRAM Read Request Queues and transfers those requests with data to the SDRAM Manager. It receives read-back data from the SDRAM Manager and sends it to the Internal Memory.

Function: (SDRAM➔ Internal Memory).

1) SDRAM Read Manager sends data with BID and IMBID to the Internal Write Manager: Rd_SRRQ{top of queue} <= BID, IMBID, SDRAM_Data.
2) Request the Internal Write Manager block to write data into Internal Memory: Wr_IM{IMBID; SDRAM_data}.
3) Update CAM with new BID (send a command to the MIM): Wr_CAM{IMBID; BID, valid=1, bypass=1, readback=1}.

Internal Read Manager Block

This block receives input from the Internal Memory and manages its output to the Reassembly block and the SDRAM Write Manager.

Function 1: (Internal Memory ➔ Reassembly).

1) Look into CAM to see if BID from top of De-queue Queue matches with BID in CAM. Issue a command to MIM: Compare_CAM{BID; IMBID, status, hit/miss}. If hit/miss = 1, get IMBID as output from CAM. Put IMBID into the output queue inside the IRM block. Assert Mem_Ras_Available to the Reassembly block.
2) The Reassembly block asserts the Ras_Mem_Cntl_Pop signal to the MM block. The IRM sends the cell control information at the top of the queue by popping from top of the De-queue Queue block.
3) At a specified time slot the Reassembly block asserts the Ras_Mem_Data_Pop signal to the MM block. The IRM starts to read data from the IM, Rd_IM{IMBID} <= Reassembly_Data.
4) Send data to the Reassembly block. Pop IMBID from the output queue.
5) *If (REL=1) or (REL=0 and "bypass" = 1 and "readback"=1)*
   Then return IMBID to Free Internal Memory List: Return_ID{IMBID}.
6) Update CAM: Wr_CAM{IMBID; BID= 00, valid=0, bypass=0,readback=0}.
7) *If (REL = 1) then*
   Release BID to FBL in PFQ: assert Mem_PFQ_Valid high, Mem_PFQ_Com = "11" (release BID) and Mem_PFQ_Param = BID.

(Refer to Multicast and BID/IMBID Release section)

Function 2: (Control data path to Reassembly and SDRAM).

Inputs and outputs of the Internal Memory are time shared as follows (to handle a dual-port 128-bit wide SRAM at 200 Mhz):

1. Slot 1 (period = 5ns): output to Reassembly.
2. Slot 2 (period = 5ns): output to SDRAM Write Manager.

Function 3: Microprocessor Access.
When switched to this mode, the microprocessor can read from the Internal Memory (IM) at a specified address. During this mode, there is only the CPU interface block that can access (read/write) to the IM block.
CPU_Rd_IM{input: Adr; output: Rd_value}: read from IM at the Adr location with the value of Rd_value.

SDRAM Write Manager Block

This block receives write requests from the SRAM Write Request Queues and then gets data from the Internal Memory to send to the SDRAM Manager block.

Operation: (Internal Memory → SDRAM).

1) Get BID and IMBID from the top of SDRAM Write Request Queues (refer to SDRAM read/write strategy section for more details on algorithm to select proper banks of memory to read and to write).
2) Look into MIM to check status of that BID: Rd_CAM{IMBID; BID, status}.
3) If "bypass" = 1,
   Then do not write data to the SDRAM. Return BID to FBL. Go to last step (5).
   Else if "bypass" =0, then get data from the Internal Memory: Rd_IM{IMBID} <= IM_Data. Send IM_Data to the SDRAM Manager along with BID and IMBID.
4) Return IMBID to the Free Internal Memory List and Update CAM:
   Return_ID{IMBID}
   Wr_CAM{IMBID; BID=00, valid=0, bypass=0,readback=0}.
5) Pop the BID and IMBID from the top of SDRAM Write Request Queues.

De-queue Request Manager Block

This block receives De-queuing requests from the Per Flow Queue block with other related information. It will look into the CAM to see if there is a CAM hit. If it is, it will modify CAM to reflect its new status, that is now a "bypass" data and should not be written into the SDRAM. It then lets the Internal Read Manager take data out of the Internal Memory and send it to the Reassembly block. If there is no CAM hit, it will send a read request to the SDRAM Read Request Queues to get data out from the SDRAM.

Function: (Per Flow Queue → Internal Memory, SDRAM).

a. Store BID, PortID and other cell control related information for De-queuing process into the end of De-queue Queue.
b. Look into CAM to see if BID is matched with BID in the CAM.
   Compare_CAM{BID; IMBID, status, hit/miss}

c. If BID is found in CAM (hit/miss = 1) and "readback" = 0 (not outgoing cell)
    If "bypass" = 0
        Then get IMBID. Modify CAM: Wr_CAM{IMBID; BID, valid=1, bypass=1, readback=0}.
    Else if no CAM hit,
        Then get free IMBID from Free Internal Memory List: Take_ID. Write BID and IMBID into end of SDRAM Read Request Queue.

Multicast and BID/IMBID Release

During Dequeuing the PFQ sends to the MM block a command to indicate whether to release the Dequeue BID to the Free Buffer List (in the PFQ) or not to release the Dequeue BID due to Multicast. Depending on the type of data inside the Internal Memory, the IMBID is also released to the Free Internal Buffer List (FIBL) in the MIM block. Figures 176-178 depicts the mechanism of the release command:

SDRAM Manager (SM)

See Figure 179. The DDR_SDRAM Controller (SM) receives the memory write (enqueue) and read (dequeue) requests from the Internal Memory Manager (IMM). All operations are done on a per controller basis independent of the other controller. Each of the two SDRAM Manager blocks, SM0 and SM1 consists of:

- RD ID FIFO
- WR ID FIFO
- BANDWIDTH OPTIMIZER BLOCK
- REQUEST SERVICE BLOCK
- CONTROLLER INTERFACE BLOCK
- BUFID ADDR DECODE BLOCK
- DENALI DDR_SDRAM MEMORY CONTROLLER BLOCK

SM0 and SM1 blocks each control corresponding pair of SDRAMs. It uses 128-bit data bus at 200 MHz in order to support 2X OC-192 transfer rate (20 Gbps). Each DDR_SDRAM Controller has the capability of peak data transfers at 16 bytes (or 128 bits) per 5ns or 25.6 Gbps for a total memory bandwidth of 51.2 Gbps in the DDR_SDRAM Manager (SM). Therefore, the DDR_SDRAM Manager must simultaneously process two memory accesses, one each to SM0 and SM1 in order to sustain the required bandwidth.

The **BANDWIDTH OPTIMIZER** block decides which of 8 possible operations, 4 reads from each bank and 4 writes to each bank to service for optimal data throughput from the SDRAMs. It looks at the current state which is the last processed request and the outstanding requests in the per bank queues and picks one based on an algorithm detailed below to optimize the data throughput from/to the SDRAMs. Some of the avoided situations include sequential same bank operations with their bank activate and precharge delays, read followed by write commands with the associated pitfall of turn around cycle delays, etc.

The Bandwidth Optimizer Block will scan through the 8 queues, 4 reads and 4 writes per Memory Manager, DM0 and DM1, and select an operation request to the appropriate bank based on the SDRAM Read/Write Strategy discussed below.

- Each series of Read requests to external memory will have to follow by a Write request to a different bank of memory from the accessed bank of the last Read.
- Within a series of Read requests, consecutive Reads should have different banks.
- Within a series of Write requests, consecutive Writes should have different banks.

For Example:

Rd(b0) → Rd(b1) → Rd(b2) → Rd(b3) → Wr(b0) → Wr(b1) → Wr(b3) ...

At the same time the read and write request BID/IMBID are stored in the **READ ID FIFO** and **WRITE ID FIFO** respectively to be returned to the IMM with the read data or a write request confirmation respectively. For write request, IMBID is returned so that IMM can update the CAM and release IMBID to Free Internal Memory List. For read request, BID/IMBID are returned so that IMM can store the read data to IM and release BID to Free Buffer List.

The **REQUEST SERVICE BLOCK** is activated with the particular request to be serviced by the Bandwidth Optimizer block. It routes the data back and forth appropriately to and from the dram and communicates with the Address Decode block and the Memory Controller Interface block. It also handles the back pressure from the Denali Memory Controller in case the command queues or the data queues in that block have reached their limit.

The **BUFFERID ADDRESS DECODE BLOCK** does the address translation of the BID to the absolute byte address and feeds this to the Memory Controller Interface block via the Request Service Block.

The **CONTROLLER INTERFACE BLOCK** formats all the commands for the Denali Memory Controller. It processes the address and data part and sends them to the Memory Controller along with required control signals in the format expected by the Memory Controller. It also receives the data from the memory controller for read operations and sends it to the Request Service Block. It will pass on control signals from the Controller to the Request Service block signaling that the controller command or data queue is full and when it is ready to receive the next request

From the above block onwards the interface of the **DENALI MEMORY CONTROLLER** and the SDRAMs is detailed in the diagram below. The Memory Controller takes in the Address and Data from the Memory Controller Interface Block and formats them into appropriate RAS and CAS commands for the DRAM memories. It also does the refresh of the SDRAM memories.

See Figure 180.

CPU Interface

Assumption: CPU access requests are infrequent and are for debugging and testing purpose only.

SM services the CPU request along with the en-queuing and de-queuing requests with equal priority. Whenever CPU requests an access to DDR-SDRAMs, SM will process the request at the end of one round scanning and servicing the 8 queues of the read/write requests. In order not to interrupt the bandwidth for enqueuing and dequeuing,

each CPU access will be serviced after X number of rounds of servicing the read and write queues. (The X number is programmable through a register)

Since the en-queuing cells and de-queuing cells are 64-byte and hence the DDR SDRAMs are programmed to have burst length of 4 (64x8 = 4x128), the CPU data aceess should be in the same cell unit, 64-byte.

Memory Description

Summary of Memories:

See Figure 181.

Summary of CAM:

See Figure 182.

Interface

Segmentation Engine

See Figure 183.

Per Flow Queue Engine

See Figure 184.

DDR-SDRAM

See Figure 185.

Reassembly

See Figure 186.

CPU Interface

See Figure 187.

Global Sync

See Figure 188.

Interface with DDR-SDRAM

The Memory Manager can support up to 512 Mbytes of external memories using DDR-SDRAM. The DDR-SDRAM device that is available at this time is the 1M x 32bit x 4banks (128 Mbits). The memory size can be programmed using register Mem_Config[2:0] (refer to the Register section).

2 DDR-SDRAM devices are required to support a data bus of 64bit. Hence 2 data strobes, Mem_SDR_DQS_Lo and Mem_SDR_DQS_Hi, are used to for the lower and upper 32bit of the data bus respectively.

I A duplication set (set 1) of memory address and control signals (including memory clock) is available for each controller in order to improve its signal driving capability when the number of devices are increasing (up to 16 devices per controller). Hence each interfacing signals will not drive more than 8 devices in any configuration of memory sizes.

This second set of address and control pins can be activated by programming the register Mem_Config[3].

Example 1:

Refer to Figure 189 for the configuration of 64 Mbytes of memory. Mem_Config[3] = 0.

Example 2:

Refer to Figure 190 for the configuration of 256 Mbytes of memory. Mem_Config[3] = 1.

Register Description

See Figure 191.

Commands

CPU_Wr_SDRAM

See Figure 192.

CPU_Rd_SDRAM

See Figure 193.

CPU_Wr_ FIML

See Figure194.

CPU_Rd_FIML

See Figure 195.

CPU_Wr_CAM

See Figure 196.

CPU_Rd_CAM

See Figure 197.

CPU_Init_MM

See Figure 198.

Principle of Operation

Initialization Setup

At power-on-reset, the entire block is reset by MEM_Reset_L signal. After power-on-reset, all registers are at default values.

CPU can issue the CPU_Init_MM command (refer to the "Register section") that will go through proper steps to initialize all internal memories inside the Memory Manager block.

Operational Flows

Normal Mode:

The normal mode is the default mode. The Memory Manager handles both the enqueuing and the dequeuing processes.

Enqueue:

Segmentation to MM:

1. Segmentation block has data available for MM block, it starts to assert Seg_Mem_Available signal.
2. MM block asserts Mem_Seg_Cntl_Pop signal to get cell control information from Segmentation block.
3. At a specified time slot the Segmentation block sends cell control information to the MM block.
4. The MM block stores cell control information in the IWM block.
5. At a specified time slot the Segmentation block starts to send data to the MM block through the Data bus (128 bits) in 4 clock cycles at 10 ns period (every other clock tick of the 200 Mhz clock).
6. The IWM block stores data inside the IWM FIFO (4 64-byte cells).
7. After receiving the free BID from PFQ (see below), the IWM block write appropriate data from the IWM FIFO into the Internal Memory (IM).

MM to PFQ:

The IWM block sends cell control information to the PFQ block on the Mem_PFQ_Param along with a command via the Mem_PFQ_Com bus. The Mem_PFQ_Valid signal is asserted high to let the PFQ block knows that the data bus and command bus contains valid data. If the Discard signal is not asserted then the Mem_PFQ_Com value should be 01 (enqueue). . If the Discard signal is asserted then the Mem_PFQ_Com value should be 10 (discard).

1. If it is an enqueue command, the PFQ block sends back to the MM block a BID at a specified number of clocks later via the PFQ_Mem_Param bus.This BID is used as an address to write data into the external SDRAM. The PFQ_Mem_Valid signal is asserted and the PFQ_Mem_Com value is 01.
2. IWM block receives BID and tags it along with the associated data being stored in the IWM FIFO.

3. If it is a discard command, the PFQ block sends back a BID with value of "7FFFFF" to the MM block at that specified time slot. In this case, data from the Write FIFO will not be written inside the external SDRAM (it will be flushed out of the FIFO).
4. If the PFQ decides to drop the cell (even when the command is not a discard), it will send to the MM block a BID with value of "7FFFFF". In this case, data from the Write FIFO will not be written inside the external SDRAM (it will be flushed out of the FIFO).

MM to SDRAM:

1. Data from the Segmentation is temporarily stored in the IWM FIFO.
2. After receiving a free and valid BID from the PFQ, the IWM block requests a free IMBID from the Memory ID Manager (MIM) block. The CAM is also updated properly with the new IMBID and BID.
3. IWM writes data from the IWM FIFO to the Internal Memory (IM) block at address IMBID.
4. If the cell is not a "bypass" cell, then IWM sends BID and IMBID to the SDRAM Write Request Queue (SWRQ) block. Place the information at the bottom of the queue that associates with memory bank of the BID address.
5. The SDRAM Write Manager (SWM) block goes through the "Write Bank Selection Algorithm" and it selects a BID/IMBID pair from a particular queue inside the SWRQ block. The SWRQ sends that IMBID to the MIM block for status verification.
6. If that cell is marked as a "bypass" cell (this means that it is being dequeued and ready to go out to the Reassembly block), then do not write data to the SDRAM. Pop the BID/IMBID from the top of the queue inside the SWRQ block.
7. If that cell is not valid as indicated in the status (this means that it has gone out to the Reassembly block), then do not write data to the SDRAM. Pop the BID/IMBID from the top of the queue inside the SWRQ block.
8. If that cell is marked as a "non-bypass" cell, then
   a). The SWM block sends request to the IRM block with BID/IMBID.
   b). IRM reads data from the IM block (Rd_IM{IMBID}) and send this data to the SDRAM Manager block along with BID/IMBID.
   c). Return IMBID to the MIM block and update its CAM:
   - Return_ID{IMBID}
   - Wr_CAM{IMBID; BID=00, valid=0, bypass=0, readback=0}
   - Pop BID/IMBID from the top of the queue inside the SWRQ block

Dequeue:

PFQ to MM:

1. At a specified time slot, PFQ sends to the Read Request Queue & Manager (RRQM) block dequeued cell control information that includes a BID via the Data2 bus. Both Valid2 and Deq signals are asserted to let the MM block knows that the data bus is valid and contains dequeue information.
2. The RRQM block stores cell control information at the end of the queue.
3. The RRQM block sends a BID from the top of the queue with a read request to the Memory Read/Write Controller (MRWC).

SRAM to MM:

1. The MRWC block monitors the Read FIFO. If it is not full, the MRWC sends a dequeue BID to the SRAM Interface (SI) block with a read command ("10").
2. The SI block puts the BID and the command in a queue.
3. The SI block executes a read command to the external SRAM with the BID as an address. Data is read out from the external SSRAM and sends to the Read FIFO. The associated BID will be placed inside the Read FIFO along with the data.
4. When a complete cell is read back from the SSRAM, the Read FIFO sends its status and the associated BID to the MRWC block.
5. The MRWC block sends the BID back to the PFQ block via the MM_PFQ_Data bus. The Valid is asserted and the Com value is "11" (dequeue).
6. PFQ receives the BID and releases it back to the Free Buffer List block.

MM to Reassembly:

1. The Read Request Queue & Manager (RRQM) block monitors the Read FIFO status.
2. Whenever there is a complete cell available inside the Read FIFO, the RRQM sends the MM_Ras_Available to the Reassembly block.
3. The Reassembly block asserts the Cntl_Pop signal to the MM block. The RRQM block sends the cell control information at the top of the queue to the Reassembly block at a specified time slot.
4. At a specified time slot the Reassembly block asserts the Data_Pop signal to the MM block. The Read FIFO starts to send data out to the Reassembly block at a specified time slot via the 128-bit Data bus at 10 ns period (every other clock tick of the 200 Mhz clock).

Flow charts for Functional Operations

See Figures 199-205.

Reassembly

The Reassembly Engine is a state machine that reassembles the stored switch cells into outgoing ATM, AAL-5, packets, or switch cells.

Main Features

- Can provide a maximum of 1M total flows.
- Uses PER PORT queues for re-assembly.
- Reassembles segmented cells back into IP packets. These packets were segmented into cells for switching at the ingress side and need to be reassembled back
- Maximum packet size of 64 KB can be handled
- AAL5 trailer field check for length and CRC-32 for all flows that have to be re-assembled and sent out as packets.
- L2 CRC check for packets that have the L2 CRC check enabled.
- Can strip up to 16 bytes of Header data.
- Reassembles 48 byte AAL5 cells or 52 byte ATM cells, both stored as 64 byte memory cells into outgoing packets, ATM cells or switch cells.
- No lookup, pass-through traffic
- For ATM
  - o VPI, VCI header translation
  - o Operation, Administration and Maintenance (OAM), Cell Loss Priority (CLP) and Explicit Forward Congestion Indication (EFCI) replacement
  - o Mark End of Packet in Payload Type Indicator (PTI) Field for AAL5 Egress Cells
  - o Optional MPLS tag append
  - o Encapsulate raw cells into packets for routing through packet fabrics.
  - o Encapsulate ATM cells into packets for routing through packet fabric.
- MPLS lookup
  - o POS (PPP/IP) packet format
  - o MPLS push/pop
  - o MPLS header translation
  - o Decrement TTL and drop if TTL is one before decrementing
- It has one re-assembly going on for each of the ports that are conFigured and active, supports a maximum of 64 ports plus a CPU port.
- The output of this block goes to the PHY PORT.

- Supports the traffic represented in Figure 206.
- Applications for various traffic types
- Types of reassembly:
  - TYPE 1: Ingress TM (ATM, AAL5, Packets) to 64B Switch Cells
  - TYPE 2: Ingress ATM cells to Packet Data
  - TYPE 3: Ingress AAL5 cells to Packet Data
  - TYPE 4: Ingress Packets to Packet Data
  - TYPE 5: Egress (ATM) to ATM cells with Translation
  - TYPE 6: Egress (AAL5) to Packet Data
  - TYPE 7: Egress (Packets) to AAL5 Cells
  - TYPE 8: Egress (Packets) to Packet Data

- Calculates and checks the CRC-32 for AAL-5 reassembly packets
- Each port has a 32-bit counter to count packets/cells transmitting from that port
- Each port has a 48-bit counter to count number of data bytes transmitted through that port
- Each port has a 16-bit counter to count packet CRC errors.
- Each port has a 16-bit counter to count TTL Timeout errors.
- Internal memory to store control and data for the 64 ports
  - Dual-port SRAM
    - CPU Data - 32 x 64 bits to store CPU data
    - Data - 1024 x 64 bits to store Cell data
    - Header - 256 x 64 bits to store Header data
    - Output Port Control - 128 x 21 bits to store Output Port Control information and Next Pointer links.
    - Output Port Queue - 64 x 24 bits to store Output Port Queue Pointers to the Data cells
    - CRC - 64 x 32 bits to store the partial CRC for each port
  - Single-port SRAM
    - Reassembly Table - 64 x 18 bits to store packet length, SOP and EOP
    - Free Buffer - 128 x 10 bits to maintain the free buffer list for data SRAM
    - Statistic - 64 x 96 bits to store packets, data and errors received for each port
- Operates at 200 MHz

Functional Description

See Figure 207. The Reassembly Engine reassembles the outgoing data from cells stored internally into ATM cells, Switch cells, or Packet data based on the application.

As shown in Figure 206 there could be 8 different types of reassembly. The operation of the Reassembly Engine for each of these types is explained in detail in the following sections.

TYPE 1 Reassembly - Ingress Memory cells => Switch cells

Type 1 reassembly maps the 64-byte cells in Memory into switch cells for transmission by adding a Switch Header. This Reassembly type handles Application types 0-3 & 7. Figure 2 shows the one to one mapping of ATM Memory cells into switch cells. The switch header will be 8 or 16 bytes long with the required cell fabric header being first and our 4-byte header following with the remaining bytes being Pad. Type 1 Reassembly also informs the SPI interface of the number of valid bytes in each 8-byte word and which Cell is the SOP and which Cell is the EOP for Packet traffic. For AAL5 cells the cells were was stored in memory as AAL5 switch cells with the AAL5 trailer at the end and Pad to fill the cell to 64 Bytes. For Packet traffic the cells were stored in memory as switch cells with an AAL5 like trailer added to pass on the packet length information. Figure 3 shows the AAL5 encapsulated cells and Figure 4 shows the pseudo AAL5 encapsulated packets. For Ingress TM types 0 & 1 that transfer ATM cells the OAM, EFCI and CLP bits will be inserted in our 4 byte switch header but for Ingress TM packets, types 2 & 3, the bits will be set to zero.
See Figures 208-10.

TYPE 2 Reassembly - 52 Byte ATM cells => Packet Data

Type 2 Reassembly maps the 52-byte ATM cells stored as 64 byte cells in Memory into packets and adds a packet header before sending it on to the SPI interface. Only one ATM cell will be mapped into each packet. Eight of the 12 bytes of Pad data at the end of each cell will be removed before the packet is sent out. This will allow us to send out a 64-byte packet including 8-bytes of header. If the Switch Packet Header is 16-bytes then the packet will be 72-bytes long. The OAM, EFCI and CLP bits will be passed in our 4 bytes of the Switch Packet Header.

See Figure 211.

TYPE 3 Reassembly - 48 Byte ATM cells => Packet Data

Type 3 Reassembly maps the 48-byte ATM cells stored in Memory as 64 byte cells into packets by stripping the 16 bytes of PAD from each cell and then optionally stripping the L2 Packet Header and adding a Switch Packet Header before sending it on to the SPI interface. It must check the L2 CRC to ensure the packets integrity before sending on to the switch fabric as it also checks the CRC in the AAL5 Trailer and removes the AAL5 encapsulation. The OAM, EFCI and CLP bits will be passed in our 4 bytes of the Switch Packet Header. It must check the L2 CRC to ensure the packets integrity before sending it to the switch fabric as the L2 CRC being sent out is now incorrect due to the removal of the L2 Header and must be replaced by the Egress chip when it adds a new L2 Packet Header. The Lookup engine will replace the L2 Header and the Segmentation will calculate the new CRC.

See Figure 212.

TYPE 4 Reassembly - Switch cells => Packet Data

Type 4 Reassembly maps the AAL5 like Switch cells stored in memory into packets removes the AAL5 encapsulation and may add an 8 or 16-byte Switch Packet header. This can also operate without a lookup operation. (Traffic Shaper, Packet in, Packet out)

See Figure 213.

TYPE 5 Reassembly - 52 Byte ATM cells + PAD => ATM cells with Translation

Type 5 Reassembly translates the ATM Header by replacing the 28-bit VPI/VCI then replaces the OAM, CLP & EFCI bits before the cell is sent out. The extra 12 Bytes of Pad data from the end of each ATM cell stored as 64-byte cells in Memory is stripped before sending it out as a 52-byte ATM cell. Using the data byte enables of the SPI interface strips the extra Pad.

See Figure 214.

TYPE 6 Reassembly - 48 Byte ATM cells => Packet Data

Type 6 Reassembly maps the 48-byte AAL5 encapsulated ATM cells stored in Memory as 64-byte cells into packets by stripping the 16 bytes of PAD from each cell and then performing a MPLS operation based on the contents of the Flow ID look-up table before sending it on to the SPI interface. It also checks the CRC in the AAL5 Trailer before removing the AAL5 encapsulation. If the MPLS TTL is one before being decremented the packet will be dropped in Reassembly.

See Figure 215.

TYPE 7 Reassembly - 48 Byte ATM cells => AAL5 cells

Type 7 reassembly adds the 4 byte ATM Header along with 4 bytes of PAD, so it is 8 byte word aligned, to the 48-byte AAL5 encapsulated ATM cells stored in Memory as 64-byte cells, strips the 16 bytes of Pad, replaces the OAM, CLP & EFCI bits in the ATM Header and then transmits them as AAL-5 cells. The 4 bytes of alignment PAD will be stripped using the byte enables when transmitting to the SPI. The EOP AAL-5 cell is marked using the PTI field in the ATM Header.

See Figure 216.

TYPE 8 Reassembly - Switch cells => Packet Data

Type 8 Reassembly maps the AAL5 like Switch cells stored in memory into packets and performs a MPLS operation based on the contents of the Flow ID look-up table. It also checks the CRC in the AAL5 Trailer before removing the AAL5 encapsulation. If the MPLS TTL is one before being decremented the packet will be dropped in Reassembly.

See Figure 217.

Block Descriptions

See Figure 218.

Description of INPUT/ENQUEUE_SM Block

The CTRL_SM block receives PID, FID, TYPE, EOP, SOP, CLP, EFCI and MEM_RAS_AVAILABLE from the Memory Manager. When AVAILABLE is asserted the CTRL_SM pops the control information from the Memory Manager control FIFO. The state machine then reads from the control SRAM using PID as the address to retrieve parameters for that port. The control state machine also requests a pointer to a free buffer from the Free Buffer List (FBL). Then it loads the parameters into the internal registers and starts the DATA_SM. The type of Reassembly is decoded from the TYPE field by the control state machine and the enable signal for each type is passed to the data state machine. The SOP and EOP signals inform the Reassembly engine when the first and last cells of a packet are received.

The FID is used to read the external Header Look-Up SRAM to retrieve the appropriate header and MPLS information for that Flow Id. This Header is passed on to the DATA_SM to be added to the reassembled cell.

The PID and other control signals are read from the Memory Manager by the Reassembly engine 1 clock before the data to give the control state machine enough time to read the per port parameters from the SRAM and load into internal registers, retrieve a free buffer pointer from the FBL and to start the header read from external memory.

Description of OUTPUT/DEQUEUE_SM Block

The data state machine starts when it receives the start signal from the control state machine. Once started, the data state machine reads 64-bytes of data from the Memory Manager data FIFO. The data state machine will also determine from the TYPE field whether to check the CRC.

When a complete cell is read and the data state machine has added the appropriate header information obtained from the external Look-Up SRAM, the data state machine informs the control state machine that it is finished reading the data for that cell. The control state machine then stores the per-port parameters back in the reassembly control table. The data state machine is then ready to receive more data.

"Once the next cell has been received and determined not to be an EOP cell then the previous cell for that port is queued into the Q_FIFO for transmission to the SPI. If the next cell is an EOP then the whole cell must be read and the length determined from

the AAL5 trailer to ensure the proper amount of valid data in the previous cell before it is queued for transmission.

Once the data for a cell has been stored and the control information is queued in the Q_FIFO, the Reassembly asserts READY to the SPI interface so it can come and Pop the Q_FIFO for the control information for that cell. Once the Q_FIFO has been Popped by the SPI, the Reassembly will start reading the data associated with that control information and send it on to the SPI 64 bits per 200 Mhz clock cycle until the complete cell is transmitted. A Data_Valid signal will be active when the data is valid on the bus and 8 byte enable signals will inform the SPI which bytes have valid data in each 64 bit transfer. This operation will allow the SPI to control the flow of data by controlling when the Q_FIFO is Popped and also strip the extra pad bytes when necessary.

The data state machine has two types of counters, one for operation and one for statistic. The cell byte count and packet length counters are for operation. The packet received count and data received count and CRC errors are for statistic. All counters are per port basis.

Description of CPU I/F Block

This is a standardized interface to the CPU in the system. It will be instantiated within each block that requires read/write access to configuration registers or internal and external memory arrays. During system operation the reassembly engine will use this interface for writing configuration registers and the header lookup table in external SSRAM as well as to receive status and statistic information. The CPU will also receive Data Cells or Packets through this interface using the CPU Port Buffer (port 65).

Description of CPU_PORT Buffer Block

This is a 32 x 64 bit internal Dual Port SRAM to store the incoming data on a per memory cell basis such that the Reassembly Engine can reassemble the data into the proper cell configuration before sending it to the CPU. The memory is organized as 4 64-byte buffers to allow 4 cells for the CPU port to be stored.

Description of REASM_TABLE_SRAM Block

This is a 64 x 18 bit dual-port internal SRAM to store the parameters for the 64 ports. This RAM is accessed twice (1 read and 1 write) by the control state machine for each data cell that is received for a particular port.

Description of FBL Block

See Figure 219. This is a 128 x 10 bit single-port internal SRAM to store the free pointers to the cells in the data SRAM. This list needs to be initialized by software after reset to have the free pointers. The control state machine requests a free pointer by reading the list, and the data state machine returns a pointer by writing back to the end of the list when the SPI receives a complete cell from the Reassembly block.

Description of CRC_GEN Block

This block generates the CRC that covers the PDU (Protocol Data Unit) in the AAL-5 format. The polynomial used is CRC-32 as shown in Figure 220. There is a minimum of 3 clocks delay in generating the CRC because the internal data path is 64

bits and the CRC polynomial is calculated 32 bits at a time. The CRC is generated in 2 stages, first the upper 32 bits and then the lower 32 bits.

The reassembly engine calculates CRC-32 for the AAL-5 data. The CRC_GEN block reads the partial CRC with port ID as the address to the CRC SRAM. The CRC_GEN block loads the partial CRC into the CRC engine and starts calculating as the data is coming in from the memory manager. When the complete cell is loaded in the internal SRAM the CRC_GEN block writes back the partial CRC into the CRC SRAM, again using port ID as the address. At the end of a packet, the CRC is compared to the CRC in the AAL-5 trailer and the packet is marked as bad when sent to the SPI out interface if the compare fails.

Description of CRC_TBL Block

This is a 64 x 32 bit dual-port internal SRAM to store the partial CRC for each port. The data state machine controls the read or write to this table based on the traffic type.

Description of DATA_PIPE Block

The DATA_PIPE block delays the data to the Data SRAM so that it matches the delay to lookup and inserts the appropriate header information and generates and check the CRC-32 for all Packet reassembly.

Description of DATA_SRAM Block

This is a 1024 x 64 bit internal Dual Port SRAM to store the incoming data on a per memory cell basis such that the Reassembly Engine can reassemble the data into the proper cell configuration before sending it to the SPI. The memory is organized as 128 64-byte buffers to allow 2 cells for each port to be stored.

Description of HEADER_SRAM Block

This is a 256 x 64 bit internal Dual Port SRAM to store the Header data on a per memory cell basis such that the Reassembly Engine can reassemble the data into the proper cell configuration and attach a programmable Header to it before sending it to the SPI. The memory is organized as 128 16-byte buffers to allow 2 16-byte Headers for each port to be stored.

Description of OUTPUT PORT QUEUE Block

This is a 64 x 30 bit single-port internal SRAM to queue information of the complete cells. The data state machine queues the cell control information when a complete cell is ready to send to the SPI interface. The Reassembly Engine sends an Available signal to the SPIO when the Q_FIFO is not empty telling the SPIO that at least one cell of data is available for transmission.

Control information delineates the SOP, EOP and SOB as well as the Port Number and whether the packet data had a CRC error or another error condition occurred. The FIFO also holds the 10-bit pointer to the data cell so the Reassembly will send the associated data with that control information.

Description of OUTPUT CONTROL_SRAM Block

This is a 64 x 30 bit single-port internal SRAM to queue information of the complete cells. The data state machine queues the cell control information when a complete cell is ready to send to the SPI interface. The Reassembly Engine sends an Available signal to the SPIO when the Q_FIFO is not empty telling the SPIO that at least one cell of data is available for transmission.

Control information delineates the SOP, EOP and SOB as well as the Port Number and whether the packet data had a CRC error or another error condition occurred. The FIFO also holds the 10-bit pointer to the data cell so the Reassembly will send the associated data with that control information.

Description of External Look-Up Memory

This is a 2M x 72 bit external SRAM to store the Header control information per FLOWID. This includes the appropriately programmed new header for the particular FLOW or the necessary MPLS manipulation control information for that FLOW. This can handle 1M FLOWs with 16 Bytes of header and 2 bytes of control information per FLOW.

Description of MPLS Label Manipulation

The FLOW-ID Lookup table will contain the MPLS label operation instructions and new labels. Three bits in the first control word will contain the encoded instructions for what MPLS operation to perform. The operations will be either replace the MPLS tag, Push the MPLS tag once or twice or Pop the MPLS tag once or twice. After each MPLS operation the original MPLS TTL value will be decremented. If the TTL value is one before being decremented then the packet must be dropped. This will be done by not enqueuing the packet into the output Queue FIFO and subsequently dropping the following cells from the FLOW until the EOP is reached.

If two labels are being pushed then Label 1 will be pushed on first and Label 2 will be pushed on second. If one label is pushed label 1 will be used. If two labels are popped the third label will be the one remaining on the top and if one label is popped then the second label will be remaining on top.

The TTL will be decremented as follows:

- Replace MPLS Tag – Decrement MPLS TTL
- Push 1 MPLS Tag – Decrement original MPLS TTL
- Push 2 MPLS Tags – Decrement original MPLS TTL
- Pop 1 MPLS Tag – Decrement original MPLS TTL
- Pop 2 MPLS Tags – Decrement original MPLS TTL

TAGS

Figures 221 and 222 show the data formats for each of the supported protocols. The specifications are not hard coded. The port table will be used for the header's location. Each field is 8 bits wide with 12 bits VPI and 16 bits VCI field, for a total of 28 bits lookup tag. This 28 bit lookup tag is located in the 1st 64-bit word, bits 32 thru 63. The PPP protocol is part of the link layer header. The tag is bit 60:41 of the first 64 bits word.

ATM TYPES

If the type of the traffic is ATM, then the CLP, EFCI, and OAM bits are included in the Switch Header for Ingress operation and inserted into the ATM Header for Egress operation. See Figure 223.

Description of Memories

There is one external Memory interface and 10 internal memory blocks for the Reassembly Engine. There is a 64 x 18 bit single port SRAM for a Reassembly Table SRAM, a 128 x 18 bit dual-port SRAM for Output Port Control information, a 64 x 18 bit dual-port SRAM for Output Port Queue Pointer information, a 128 x 7 bit single port SRAM for a Free Buffer list, a 1024 x 64 bit dual-port SRAM for DATA SRAM, a 256 x 64 bit dual-port SRAM for HEADER DATA, a 36 x 64 bit dual-port SRAM for CPU DATA, two 64 x 32 bit dual-port SRAMs for CRC Tables and a 64 x 112 bit single port SRAM for Statistic information.

Data Structure for Reassembly Table

See Figure 224. The REASSEMBLY_TABLE is a 64 x 18 bit internal SRAM addressed by the port ID (PID). The Reassembly Table has 64 entries, one for each port.

- SOP: Start of Packet
- EOP: End of Packet
- Packet Length: A running count of the number of bytes in that packet.

Data Structure for OUTPUT_CONTROL_MEMORY

See Figure 225. The OUTPUT_CONTROL_MEMORY is a 128 x 18 bit internal SRAM that is used to store the control information for each cell that is ready to send to the SPI block along with the next read pointer and next number of Header words for the next cell linked to that port in the DATA SRAM and the number of valid bytes in that cell.

- SOP: Start of Packet
- EOP: End of Packet
- CELL_LENGTH: Number of Data bytes valid for that cell. Used internally to create the Byte valid signals to the SPI interface. (Combine with number of Header words for RAS_SPI_CELL_LENGTH sent to SPI OUT.)
- NEXT RD_PTR: Pointer to the Head of the next cell of data queued to that port in the Data SRAM.
- NEXT HDR_WD: Number of header bytes for the next cell of data queued to that port in the Data SRAM.

Data Structure for OUTPUT_PORT_QUEUE

See Figure 226. The OUTPUT_PORT_QUEUE is a 64 x 18 bit internal SRAM that is used to store the output port Queue Head (RD_PTR) and Queue Tail (WR_PTR) pointers for each cell that is ready to send to the SPI block along with the port Empty status and the number of Header words in the Queue Head cell.

- RD_PTR: Pointer to the Head of the next cell for that port that should be sent to the SPI Out.
- WR_PTR: Pointer to the Head of the last cell for that port that should be sent to the SPI Out.
- E: Empty – Status of cells queued and linked for that port. (Default = 1, Empty)
- PEND: Cell Pending – Cell is waiting to be queued to output port. (Default = 0, Not Pending)
- HDR_WDS: Number of header words for the next cell for that port to be sent to the SPI Out.
-

Data Structure for FREE_BUFFER

See Figure 227. The FREE_BUFFER is a 128 x 7 bit internal SRAM that is used to store the pointer to the free data block in the DATA SRAM. A pointer is taken from the top of this table when a data block is needed to receive the incoming data. When the SPI interface gets a complete cell off the queue the pointer to that cell is returned to the end of this table.

- FREE PTR: Pointer to the head of a 64-byte buffer in the Data SRAM.

Data Structure for DATA_SRAM

See Figure 228. The DATA_SRAM is used to store the data while reassembling it into the proper cells for transmission. There are 128 64-byte buffers. This can support 128 64-byte cells.

Data Structure for HEADER_SRAM

See Figure 229. The DATA_SRAM is used to store the data while reassembling it into the proper cells for transmission. There are 128 16-byte buffers. This can support 128 16-byte Headers.

Data Structure for CPU_DATA_SRAM

See Figure 230. The CPU_DATA_SRAM is used to store cell data destined for the CPU. There are 4 72-byte buffers. This can support 4 64-byte Cells of Data plus 8 bytes of control information.

Data Structure for CRC_TABLE

See Figure 231. The CRC_TABLE is a 64 x 32 bit internal dual port SRAM that stores the partial CRC for each port reassembly.

- PART_CRC: Partial CRC stored after each cell for that port. Used to calculate the CRC for an entire packet.

Data Structure for STATISTICS

See Figure 232. The Statistic SRAM is a 64 x 112 bit internal SRAM addressed by the port ID (PID). Each port has an entry to this SRAM to store the number of packets and data received for that port along with the number of CRC error packets. The content of this SRAM is accessed through the PCI interface block.

- o PKT_RCVD: Counter for the number of Packets received.
- o DATA_RCVD: Counter for the number of Data bytes received. Only includes valid data bytes.
- o CRC_ERRORS: Counter for the number of CRC Errors received.
- o TTL_ERRORS: Counter for the number of TTL Timeout Errors.

Data Structure for EXTERNAL LOOK-UP MEMORY

See Figure 233. The External Look-Up SRAM is 4M x 36 bits made up of eight 512K x 36 bit external SRAMs addressed by the FLOW ID (FID). Each FLOW has two entries to this SRAM to store the Header data for that Flow. The Flow Header Data (FHD) is two or four 36-bit words that contain up to 2 MPLS tags, 1 VPI/VCI label, or a Layer 2 Header. On the Ingress chip a total of 12 bytes are available for new user defined headers with the remaining 4 bytes of Header data used for a Maximus internally defined Switch Header. On the Egress chip all 16 bytes are available for new user defined headers as our Maximus 4 byte header will not be append in this case. The two control bytes contain a number of fields that indicate how to process the packet or cell. Each FHD is linked to a user defined Header, an MPLS tag or a VPI/VCI label through the FLOWID.

FLOW HEADER FORMAT:

See Figure 234. The following fields are in the Flow Header Data Table:

1. 1 Byte Control word in 1st FLOWID word to determines the type of FHD (MPLS, ATM or L2) and the MPLS operation to perform as well as the number of Header words.
2. 32 ATM/32 MPLS bit New Label 1
3. 32-bit MPLS New Label 2
4. 1 Byte Control word in 2nd FLOWID word to determine the number of L2 Header bytes to be stripped.
5. 32 bits for possible Longer Header Formats.
6. 32 bits for our internal Switch Header information.

CONTROL BYTE FORMAT:

See Figure 235. The first control word is bits 71 to 64 of the first 72-bit word of the FLOW ID with the second control word being bits 71 to 64 of the second 72-bit word for that FLOW ID. The following fields are in the Flow Header Control Words:

1. HDR TYPE = 2 bits that determines the type of Header (MPLS, ATM or Ethernet)
   a. 00 = MPLS
   b. 10 = ATM
   c. 01 = L2
   d. 11 = Reserved
2. MPLS OPERATION = 3 bits to indicate the type of MPLS processing to perform.
   e. 000 = Replace MPLS Tag; Decrement MPLS TTL
   f. 001 = Push 1 MPLS Tag; Decrement original MPLS TTL
   g. 010 = Push 2 MPLS Tags; Decrement original MPLS TTL

h. 011 = Pop 1 MPLS Tag; Decrement original MPLS TTL
i. 100 = Pop 2 MPLS Tags; Decrement original MPLS TTL
j. The last 3 cases are reserved for future implementations.

3. HDR = 2 bits to indicate the number of valid Header words. (00 = 0, 01=1, 1X=2)
4. L2EN = L2 CRC checking Enabled
5. # BYTES L2 HDR STRIP = 4 bits for Number of L2 Header Bytes to strip if type is ETHERNET.
6. RSVD = 4 bits Reserved for future use.

MAXIMUS 4 BYTE HEADER FORMAT:

See Figure 236. The Ingress Maximus chip will append a proprietary 4-byte header onto the end of any 8 or 16 byte switch or packet header before sending the data out. This allows the user to append a programmable 4 to 12 byte header along with the Maximus 4-byte header onto every packet or switch cell that is transmitted from the Ingress Maximus chip. The Maximus header contains information of Flow ID, application types and Class, along with the OAM, CLP and EFCI bits for ATM cells.

- FID: FLOW ID
- TYPE: Application Type
- EFCI: Explicit Forward Congestion Indication
- CLP: Cell Loss Priority
- OAM: Operation, Administration and Maintenance
- CLASS: Class of service
- EOP: End Of Packet
- SOP: Start Of Packet

The FID and CLASS will be obtained from the Header Lookup memory while the remaining signals will be obtained from the Memory Manager interface.

ATM HEADER FORMAT:

See Figure 237.

- VPI: Virtual Path Identifier
- VCI: Virtual Channel Identifier
- PT: Payload Type
  - EOP: End of Packet
  - EFCI: Explicit Forward Congestion Indication
  - MI: Management Information
- CLP: Cell Loss Priority
- HEC: Header Error Control

MPLS HEADER FORMAT:

See Figure 238.

AAL5 TRAILER FORMAT:

See Figure 239.

- CPCS-UU: Common Part Convergence Sublayer
- CPI: Common Part Indicator

- o L: Length of payload
- o CRC: Error detection in SSCS PDU (user data)

Interface

System

See Figure 240.

Reassembly ⇔ Memory Manager

See Figure 241.

Reassembly ⇔ SPI

See Figure 242.

Reassembly ⇔CPU

See Figure 243.

TIMING

See Figures 244-6.

Memories

Input phase to RAS from Internal Memory Manager. See Figure 247. Note: R2 for the Partial CRC Table is dependent on the PID. If the next cell has the same PID as the previous cell then this read information is not used.

ERROR CONDITIONS

Several errors can happen in the Reassembly engine with the error packets being marked during the EOP transfer to the SPI as EOP Abort. The different errors are listed in Figure 248.

Register Description

Registers

See Figure 249.

Commands:

See Figure 250.

Commands

Read external RAS memory

See Figure 251.

Write external RAS memory

See Figure 252.

FLOW HEADER FORMAT:

See Figure 253. The following fields are in the Flow Header Data Table:

7. 1 Byte Control word in 1st FLOWID word to determines the type of FHD (MPLS, ATM or Ethernet) and the MPLS operation to perform as well as the number of Header words.
8. 32 ATM/32 MPLS bit New Label 1
9. 32-bit MPLS New Label 2
10. 1 Byte Control word in 2nd FLOWID word to determine the number of Ethernet L2 Header bytes to be stripped.
11. 32 bits for possible Longer Header Formats.
12. 32 bits for our internal Switch Header information.

Read Internal Data Memory

See Figure 254.

Write Internal Data Memory

See Figure 255.

Read Four Internal Memories (Partial CRC1, Partial CRC2, Reassembly Table, Output Queue)

See Figure 256.

Write Four Internal Memories (Partial CRC1, Partial CRC2, Reassembly Table, Output Queue)

See Figure 257.

Read PORT Statistics

See Figure 258.

Write PORT Statistics

See Figure 259.

Read CPU Data Memory

See Figure 260.

Write CPU Data Memory

See Figure 261.

Read 2 Internal Memories (Free Buffer List (FBL), Output Control)

See Figure 262.

Write 2 Internal Memories (Free Buffer List (FBL), Output Control)

See Figure 263.

Read Internal Header Data Memory

See Figure 264.

Write Internal Header Data Memory

See Figure 265.

Get CPU Cell command

See Figure 266. This command reads 64 bytes from the CPU Data FIFO 8 bytes at a time and stores them in the general purpose registers R0- R15. There is no address associated with this command as it just reads the CPU Data FIFO 8 times and then loads the Registers with the data.

Initialize Free Buffer List and Output Port Queue Memories command

See Figure 267. This command initializes the Free Buffer List in the FBL Memory and sets the Empty and Pending bits to inactive in the Output Port Queue Memory.

Reserved Op-codes

Op-codes 17 to 31 are reserved for future use.

Principle of Operation

The re-assembly block keeps track of a FLOW ID per port. The scheduler has a weighted round robin scheme that it uses to schedule a CELL for each port. If a port has more weight (or bandwidth) then it schedules more cells for that port. These scheduled cells are provided to the de-queue engine which sends the requests on to the Internal Memory Manager for pulling them from the SRAM.

The Internal Memory Manager reads the scheduled cell from the external SRAM and stores it in an Output Data FIFO. Once the cell is stored in the Output Data FIFO the Internal Memory Manager queues the control information for that cell into the Output Control FIFO and asserts an available signal to the Reassembly Engine.

The Reassembly Engine pops the Output Control FIFO to get the control information for the available cell. It uses the FLOW ID to look up the Header processing necessary for that cell as well as the Type field to determine the type of Reassembly to perform.

It also obtains a pointer to the Internal Dual Port Data SRAM from the FBL FIFO. Once it has started the Header look up and obtained a Free Buffer Pointer the Reassembly Engine starts reading the data from the Internal Memory Manager Output Data FIFO. The data is read 128 bits at a time at 100 MHz. Each Data cell is 64 bytes long, with each data buffer being 64 bytes deep. There are 128 64-byte buffers in the Internal Data SRAM. The Header information will be stored in a separate Internal Dual Port SRAM that is also addressed from the FBL Pointer. The Header buffer has 128 16-byte buffers so it can handle up to 16 bytes per Header. The Header will be stored concurrently with the Cell Data being stored in the data buffer.

Once the Header and Data are stored locally in memory the cell control information will be queued in the Output Control Memory and the Buffer pointer will be queued in the Output Port Queue Memory. If more than one cell is queued on a port then the Next Read Pointer will be stored in the Output Control Memory of the previous cell to link the current cell to the previous cell.

The SPI has a weighted round robin scheme that it uses to Request a CELL for each port. If a port has more weight (or bandwidth) then it Requests more cells for that port. When the SPI requests a cell for a particular port the Reassembly checks the ports Output Port Queue to see if there are any cells available for that port. If there are no cells

available the Reassembly informs the SPI that the port has no data by asserting an Empty signal and the SPI continues on to the next port. If the port has cells available the Reassembly uses the Read pointer stored in the Output Port Queue to read the control information for that cell along with the associated Header and Cell Data. The Reassembly sends the Header and Cell Data to the SPI interface along with a valid signal and byte enables to specify the valid data bytes. The SPI will use the byte enables to strip any unwanted padding. After the Reassembly Engine sends the complete cell to the SPI interface it returns the Read Pointer to the FBL FIFO.

The Reassembly engine has a Reassembly status table for each of the ports. It can simultaneously reassemble one packet at a time for each port. The local port Reassembly status table stores the control information for each port along with state information and or any partial cell information. The control information tells the re-assembly block how to re-assemble the FLOW for that port, whether it is ATM to ATM, ATM to packet, packet to ATM or packet to packet.

For AAL5 to Packet reassembly a CRC- 32 is calculated on the packet as each cell is received. The partial CRC is stored on a per port basis and loaded back into the CRC engine for each subsequent cell in that packet. Once it has received the end of the packet the calculated CRC is check against the received CRC to ensure packet integrity. If the CRC is in error the packet will be marked as bad by setting the Bad control bit in the Output Queue FIFO for the SPI interface.

For Layer 2 Packet reassembly the CRC-32 is checked on the packet as it is being sent to the SPI out interface. The CRC is calculated the same as the AAL5 CRC-32, but if there is an error detected it must be sent immediately to the SPI out as the complete packet has been sent in this case. The CRC checking is user selectable by programming the CRC enable bit in the Flow ID Lookup table.

Statistics are kept on a per port basis for the number of Packets sent, the number of Bytes sent and the number of errors received. The CPU can read and write internal registers or statistics on a per port basis as well as access all internal memory locations and the external Header Lookup memory through the PCI interface. Data packets can also be sent to the CPU through the PCI interface.

Initialization Setup

The initialization process consists of writing the Initialize Memories command op-code to the Command register. The Reassembly block will then initialize the Free Buffer List and clear all the Empty and Pending bits in the Output Port Queue Memory. When the initialization process is done, the command op-code will be cleared to all zeroes to inform the system CPU that the initialization process is complete.

Re-assembly flow

1. Output scheduler selects the port.
2. Priority scheduler for port decides on the FLOW ID.
3. FLOW ID is sent to the De-queue engine.
4. Output scheduler keeps the current head and other information per port.
5. De-queue engine queues the buffer pointers along with corresponding port # for Internal Memory Manager.

6. Internal Memory Manager gives a data available signal to the reassembly engine when a complete cell is in the output data FIFO in the Internal Memory Manager.
7. Internal Memory Manager sends ACK back to the scheduler
8. Scheduler releases the buffer from the INTERNAL MEMORY MANAGER and passes next port and buffer head to the de-queue engine.
9. Internal Memory Manager tells Reassembly that a cell is available.
10. Reassembly engine pulls the control information for that cell from the Internal Memory Manager control FIFO and then extracts the data from the Internal Memory Manager output data FIFO.
11. Reassembly uses the control information to look up the Header processing necessary for that FLOW. It also obtains a free buffer pointer to the Data SRAM and uses the control information to determine the type of processing to perform.
12. Reassembly engine stores the new Header information along with the data cell and Queues the cell onto the appropriate output port.
13. The SPIO requests data on a per port basis based on its own internal port calendar.
14. Once the SPIO requests data for a port that has a cell queued on that port the Reassembly sends the control information for that cell along with the associated Header and Cell Data for transmission.
15. The SPIO will strip extraneous pad data based on the byte enables that go with the data.
16. The Reassembly engine sends CPU Buffer Full status information to the SPIO for combining with the port FIFO status information that is sent to the Scheduler from the SPIO.

Per Flow Queue Engine

The Per flow queue stores flows parameters and statistics in dedicated FID memory. In addition, it stores memory buffer pointers or IDs that hold data traffic of the flow IDs in a dedicated BID memory. Furthermore, it will retrieve the above FID parameters, statistics and memory buffer IDs when they are needed. It manages the free buffers list. It performs Random early discards on FID data cells.

Features

- Supports 1M possible flows
- Supports 1K Tunnel Flows.
- Supports 1K Multicast Flows.
- Multicast one flow to 1 K flows maximum.
- Collects statistics on a per flow basis.
    - Packet Counters: 32 Bits.
    - Byte Counters: 48 Bits.
    - Packet Discards per external Requests such as SPI: 16 Bits
    - Packet Discards per Reassembly Timeout request: 16 bits.
    - Packet Discards per internal buffer management request: 16 bits.
- Discards by returning buffer Ids to free buffer list.
- Manages the Free Buffer List.
- Manages RED by FID association.
- RED Memory: 1K x 48 bits.
- FID Enqueue Memory: 2M x 72 bits.
- FID Dequeue Memory: 1M x 36 bits.
- FID Statistics Memory: 2M x 72 bits.
- BIDLL Memory: 8M x 36 bits, including free buffer list.
- Tunneling Dequeue Leaf Memory: 1K x 56 bits, internal.
- Tunneling Enqueue Leaf Memory: 2K x 82 bits, internal.
- Tunneling Root Memory: 512 x 21 bits, internal.
- Multicast Dequeue Leaf Memory: 1K x 36 bits, internal.
- Multicast Enqueue Leaf Memory: 2K x 82 bits, internal.
- Multicast Root Memory: 512 x 30 bits, internal.
- Clock Frequency: 200 MHz.

Functional Description

See Figure 278. The Per Flow Queue Engine consists of the following blocks: Enqueue engine, Dequeue engine, Free List buffers manager, RED manager, Statistics engine, Timeout manager, and CIH Director. Enqueue engine places received data cell pointers on the queues. Dequeue engine retrieves and removes data cell pointers from the queues. The RED manager will track the total number of buffers on eight CLASS counters using each of the associated RED parameters. The cells will be dropped when the RED algorithms flags the cells as such. The statistics manager counts per flow various parameters. The timeout manager discards packets or cells when they exceed a certain timeout threshold on the enqueue side for the reassembly timeout. The timeout

manager discards packets or cells when they exceed a certain timeout threshold on the dequeue side for the reassembly timeout. The PFQ interfaces to the memory manager, to the shaper and to the scheduler. The Per Flow Queue Engine handles the flow tunneling, discarding and multicasting of FID in addition to enqueueing and dequeueing the FIDS. The statistics block will collect on a per flow basis the number of discarded flows, the number of bad flows, and the number of dropped flows because of RED. It will provide free buffers BIDS to the memory manager and it returns the buffer BIDS to the free buffer list.

Block Description

Algorithms

Enqueue Normal FID Algorithm

The enqueue request will place a BID of an FID on the queue. It will modify the BID as well as the FID Enqueue memory.

When the queue is empty, the first enqueue is for an SOP cell. If the cell is not SOP, then the cells are dropped until the next SOP. See Figure 279.

1. The FID Enqueue memory is read, the tail pointer will point to the last BID in the link.
2. The BID tail is compared to "Null"; if it is a null then the queue is empty.
3. The current BID tail that was stored in the FID Enqueue memory BIDC will be used to access the BID memory. The new BID Tail is written as the content of that BIDC Tail location. If the cell is SOP or EOP then the flags are written in the BID memory with the BID link.
4. The new BID replaces the tail pointer in FID Enqueue memory.
5. The new tail is linked to "NULL" in the BID memory.

- When the tail pointer is read, it will be checked to see if it is null. When it is null then the Dequeue memory has to be written with the new head and the associated flags such as the SOP, CLP, and EOP etc.
- If an EOP is received, then a shape command is issued to the shaper.
- The count is sent to the statistics block with the FID when the EOP flag is received.
- The count is sent to the shaper and then it is not saved in the FID or BID memories.

Dequeue Normal FID Algorithm

The de-queue request will remove a BID from an FID.

1. The FID Dequeue memory is read; the head BID pointer will point to the first BID in the link.

2. The current BID head that was stored in the FID Dequeue memory will be used to access the BID memory.
3. The new BID will replace the head pointer and the value is written in the FID memory. If the BID has other flags like SOP, EOP they are written at the same time as the BID head.

- The new BID is compared to Null. If it is equal to null, then the Queue is empty and the Enqueue memory for that FID has to have the tail as Null.
- A dequeue command is sent to the memory manager.

See Figure 280.

Enqueue Multicast FID Algorithm

When the queue is empty, the first enqueue is for an SOP cell. If the cell is not SOP, then the cells are dropped until the next SOP. The bit in the control register will determine the treATMent of multicast traffic. If the bit is set to treat the multicast traffic as Unicast, then the traffic is enqueued, as Unicast one. Else, the traffic is treated as described in the following section. The multicast control bit in the control register is reset. The PFQ will discard all the enqueue requests from the memory manager when an EOP cell is received for a multicast traffic. When the SOP cell is received, it is enqueued as a Unicast cell. The buffer ID pointers are linked in the memory. The pointers in memory are enqueued only once. The count is read from the Multicast Root memory. Every BID will have the count of the previous BID. When an SOP cell is enqueued, the tail pointer is in the enqueue memory. There is no link in the BID memory to the tail pointer in the enqueue FID yet. When a NULL is written to next location, the count is properly written in the correct field. When the EOP cell is received, then the BID link is performed as it was done for the previous cells. When the BID of the EOP cell is linked, the multicast enqueue operations has to link the head and tail BIDS to the various FIDS that are members of the multicast group. At this time, the enqueue engine will discard all the cells with the enqueue request from the memory manager until the multicast packet has been linked to all the FIDS in the multicast group. The multicast engine will perform the rest of the operations to enqueue the packet on the rest of the FIDS. The multicast engine will read the next FID that in the multicast group. The first FID member has already the pointers linked in its location. The multicast engine will update the head and the tail pointers in the enqueue and dequeue memories locations that correspond to the FID members if the FID queue is empty. If the FID is not empty, then the tail pointer is updated only. When the last FID member is update it, then the enqueue engine will continue to enqueue rather than discard the cells. The process is not efficient and the performance suffers considerably.

The buffer management whether it is RED or class will be performed on the first SOP when the cell is enqueued for the HEAD member of the multicast group. Thereafter there are no checks done. After receiving the EOP cell, the DBS will get the FIDS that are members of the multicast group only.

1. The packet is enqueued as Unicast one until the EOP cell.
2. When EOP cell is received, then the packet has to be linked for the rest of the FID members of the multicast group.
3. The FID root memory is read to retrieve the head FID, tail FID pointers.

4. The head FID has the packet already linked on its queue. The next FID field in the multicast group is read.
5. The next FID is compared to the tail FID to determine the last FID in the multicast group.
6. The LEAF memory of the FID in the next FID field is read to determine if the queue is empty. If the queue of the FID is not empty, then the tail pointer is updated it with the last pointer of the EOP cell. The new tail pointer is linked to the old tail pointer in the BID memory and the COUNT is written there, too.
7. If it is empty, then the tail pointer in the enqueue memory is updated it from the enqueue memory. The head pointer in the dequeue memory has to be updated it from the dequeue memory. The leaf memory is updated it with BID pointers.
8. The next FID in the FID leaf memory is compared to the tail FID in the root memory and the above operation is repeated until the next FID and the tail FID are equal. Then it is the end of the multicast group.
9. The PFQ_MEM_FULL signal is asserted to the memory manager.

Dequeue Multicast FID Algorithm

The dequeue command for a multicast FID is similar to the Unicast FID. They BID link is read from the BID memory and it will be the new head pointer. In addition the counter is read from the BID location that is the new head pointer. The counter is decremented and then written back in the same location with that BID. If the counter value is zero, then the "REL" bit in the command to the memory manager is set. Else the "REL" bit is reset and there will be no release command for that BID from the memory manager. If the BID link in the BID memory is "NULL", the queue for that FID is empty. The "NULL" has to be written back in that locations with the counter value equal to the previous count but decremented by 1.

1. The FID leaf memory is read.
2. The location read is the FID that is part of the dequeue command.
3. The head BID pointer in the FID memory will point to a location in the BID memory.
4. The location contains the next BID head pointer and the count.
5. The count is decremented by one and written into memory.
6. The new BID will replace the head pointer and the value is written in the FID leaf memory.
7. The "REL" bit in the command is set when the count is zero.
8. When the pointer is "NULL", it is handled the same way as in the Unicast FID.

Enqueue Tunneled FID Algorithm

When the queue is empty, the first enqueue is for an SOP cell. If the cell is not SOP, then the cells are dropped until the next SOP. The enqueue of the tunneled FID is exactly as the normal FID. When an SOP cell is received, the tail and head pointers are updated it as usual. When the EOP cell is received, then the FID that the PFQ sends to the DBS is not the same FID that was used to enqueue the packet. Instead, the FID will be read from the Leaf memory from a memory location pointed to by the FID of the enqueue command and it is in the ROOT Tunneling field. The FID that was read as the Tunnel ROOT memory represents one or more FIDS. The DBS will not know about the

members of the ROOT FID. The FID in order to be tunneled has to be a member of that specific tunnel.

1. The leaf FID memory is read, the tail pointer will point.to the last BID in the link. The root memory association is read.
2. The current BID tail that was stored in the FID memory will be used to access the BID memory. In that location the new BID is written.
3. The new BID replaces the tail pointer and the FID location is written to the leaf FID.
4. The ROOT memory corresponding location is read.
5. If the not empty flag is reset, then the LEAF FID is written as the "CURRENT FID HEAD" and as the Tail FID
6. In EOP cell cases, the ROOT FID is sent to the DBS with the cell count.

Dequeue Tunneled FID Algorithm

The dequeue command that the DBS send to the PFQ contains an FID. The FID is the ROOT FID. The root memory has to be read and the member of the tunnel group has to be found. The Root memory has the head of the FID. The head is read and the FID leaf memory is read. After reading the Leaf memory, the dequeue operation proceeds the normal way. When the cell is EOP, then the "NEXT FID" field is used to update the new FID head pointer. If the next FID queue is empty, then the next FID field in that FID entry will be used and so on.

1. The FID Root memory is read; the current FID pointer of the tunneled FID is retrieved.
2. The FID leaf memory is read and the current BID head that was stored in the FID memory location will be used to access the BID memory.
3. The new BID will replace the head pointer and the value is written in the FID leaf memory.
4. The LEAF FID and the BID are sent to the memory manager.
5. If the BID belongs to an EOP cell, then the "NEXT FID" field is read from the leaf memory.
6. The previously read "NEXT FID" field will be used to update the "CURRENT FID TUNNEL HEAD" in the root memory.
7. The new FID head leaf memory entry is read to determine if the queue is not empty.
8. If the queue of the head FID is empty, then the "NEXT FID" field of that FID is read and the process is repeated until an FID without an empty queue is reached.
9. During
10. During the process, if the tail FID is reached, then there are no more BIDS on any queue.
11. At that point the process terminates until the next dequeue command. The "NOT EMPTY" flag is reset.
12. During the above process, assert "PFQ_ DBS_FULL" to the DBS.

Enqueue Engine

See Figure 181. The enqueue engine gets its commands from the memory. The enqueue engine issues commands to the shaper. If there are no free buffers or the request

is to discard the FID, then the "NULL" FID is returned. If there is an enqueue request and there are no free buffers available, then the "NULL" BID is sent to the memory manager. If the RED determines that the packet has to be discarded, then the "NULL" BID is sent to the memory manager.

The enqueue engine will enqueue the FID, discard the BID marked for discard, enqueue the multicast FID, and enqueue the Tunneling FID. The enqueue engine has to issue the FID and the BID memory commands always within the eight cycles window. The FID of the enqueue and de-queue commands will be compared and checked if they are the same. If they are the same FID, then only the last FID memory write is valid and has the correct information. The Enqueue information will be merged with the dequeue information and written during the dequeue operation. The BID memory accesses depend on the FID results. The data coherency of the BID during multicast dequeue operations has to be maintained by the logic with respect to the BID content of that location and its count in case of multicast.

The Tunneling of FIDS as well as the multicasting of FIDS is based on a circular buffer. Once the last FID is reached, then the enqueueing engine will point to the first FID in the link. Tunneling uses the root FID in the enqueue as well as in the dequeue process. The Shaper has no knowledge of the FIDS that are members of the Tunneled FID. The shaper has no knowledge of the multicast root FID.

When the memory manager issues a command to the enqueue engine, the FID parameters are retrieved and the BID is enqueued. If the EOP is set, then a shape command is sent to the shaper. If the FID was empty, then the FID is activated.

1. The enqueue engine will read the following:
   - FID parameters from FID enqueue memory.
   - Free buffer head from the free buffer register.
   - RED counter.
2. The enqueue engine will read the following:
   - Read the BID memory location FBH to get the new FBH.
3. The corresponding statistics from statistics memory of the FID.
4. The enqueue engine will perform the checking operations of the various conditions. The counters are incremented.
5. The enqueue engine will write the following:
   - The BID memory location current BID Tail with BID as content.
   - The BID memory location new BID tail with the "NULL" as content.
   - The "FID'' enqueue location with the new parameters.
   - The "FID'' dequeue location with the new tail if the queue is empty.
   - The FID statistics location with the new statistics.
   - The RED Counter with the new value.
   - The Free buffer register with the new head.

Dequeue Engine

The Dequeue engine gets its commands from the scheduler. The Dequeue engine issues commands to the memory manager and to the enqueue engine. When there is a dequeue command, the PFQ will get all the parameters from the FID and sends a command to the memory manager for retrieval of data. Then the PFQ will update the buffers pointers

See Figure 282.

1. The dequeue engine will read the following:
   - FID parameters from the Dequeue FID memory.
   - The BID memory location BID head.
2. The de-queue engine will perform the checking operations of the various conditions. The head pointers are updated. If the head is null then the queue is empty.
3. The dequeue engine will write the following:
   - The "FID" dequeue location with the new parameters.
   - The "FID" enqueue location with the new tail "NULL" if empty.

Release BID

The free buffer engine will return the BID to the free buffer list. It can be simultaneously issued with the enqueue request. The bit "REL" will determine if the memory manager will release the BID or not.

- The free buffer BID memory with the new link BID.

Free buffer manager

There is a free buffers list. There is a corresponding head and tail registers on the device. There is a link list linked in the BID memory. The link list is initialized on init command. The enqueue will use the head of the free buffers. The tail of the free buffer list is updating it when the enqueue and dequeue engines are accessing it. The buffer number "7FFFFF" is reserved for not valid BID. The cell should be dropped when the memory manager gets this value in case of a discard request from the memory manger or the RED algorithm. See Figure 283.

There are three parameters that are stored in registers for the free buffer list.

- The number of available buffers. It represents the raw total storage memory space.
- The threshold of no buffers.
- The number of used buffers.

The following operations are done on the counters:

- Whenever a buffer is enqueued, then the number of used buffers is incremented.
- When a buffer is released, then the number of used buffers is decremented.
- The number of used buffers can never be less than "0".
- The number of used buffers can never be more than the available buffers.

When the number of used buffers is equal or larger to the threshold of number of buffers, then:

- All the queue of that cell has to be emptied and all the buffers of that queue are returned to the free buffers list. The buffers in use counter will be decremented by the number of cells.
- Update the free buffer register and memory link in the BID memory.

Timeout manager

It is running with simultaneously with the enqueue operation. The timeout manager is used to remove orphan cells for reassembly timeout and free the storage for incoming traffic. When an FID corresponding memory location is read for an enqueue

operation, there are two TTL bits that are reset to "00" when the FID memory location is written back. The memories that are accessed are the external FID memory and the internal leaf FID memories. The internal memory will be accessed once the external memory has been completely accessed once. The algorithm is not a fair one. When the timeout manager reads a location the TTL bits are changed from "00" to "01" or changed from "01" to "10". If the FID location is read and the TTL bits are "10", then there is a reassembly timeout and the packet remnants are discarded. The Qsize has to be decremented whenever there is a discard. It should have the value "0" in the statistics memory.

The statistics memory location is read. Every 8 cycles or every time slot, one location of the enqueue well as the statistics memories is read to perform the timeout operations. If there is a reassembly timeout, then the start of timeout operation is suspended until the data is updated in both the enqueue and the statistics memories. The timeout manager will do the following:

- An active address that points to an FID memory location. The address is sequential and circular. It is updated when the access is executed.
- Every sync cycle the FID memory is accessed, a location corresponding to that address is read and examined.
- If the TTL bits value is "00", then the value will be changed to "01".
- If the TTL bits value is "01", then the value will be changed to "10".
- If the TTL bits value is "10", then it is reassembly timeout and the packet chunk is discarded on the enqueue side.
- The TTL bits value then are reset to "00"after the discard.
- The DROP timeout bit is set. All the fragments of a packet has to be dropped until the next SOP. At the next SOP enqueueing starts again.

DISCARD FID

The discard operation is running in place of the enqueue operation. Whenever there is a discard because of Reassembly timeout, the Qsize has to be decremented. It should have the value "0" in the statistics memory. There are several discard commands:

- Memory manager generates a discard command of a packet that is being enqueued.
- RED manager generates a discard command of a packet violating the algorithm.
- Free Buffer manager generates a discard command of a packet being enqueued because of lack of free buffers. When the number of used buffers exceeds a programmed backpressure threshold register, the enqueue engine will signal the segmentation the above condition. If the number of used buffers drops 10 buffers below the programmed backpressure threshold, then the PFQ will signal the above message to the segmentation engine. If number of used buffers exceeds the programmed threshold register then the enqueue engine will discard these buffers until the next EOP. The PFQ will discard the packet fragments from the queue in such a condition.
- Timeout manager generates a discard command when the packet is not completely enqueued because the later chunks were not received or got dropped. The packet is a tail of the queue of that FID.

- CLASS discard; the packet will be discarded when the queue counter is above the programmed threshold. On every SOP the CLASS counters are checked.
- Port Blockage; the packet will be dropped from SOP to EOP until the bit is changed in memory. On every SOP the port register is checked.

When the memory manager generates the discard command, then the following steps are taken:

1. The FID enqueue memory is read. The previous packet tail is parsed.
2. The BID memory is read and the BID of that location is read. The content is the head of the new packet being enqueued.
3. The Free Buffer tail is read.
4. The BID memory location of address FBT will have the content the head of the packet that is being discarded. The buffers are on the free buffers list.
5. All the buffers are returned to the free buffer list.
6. The FID memory location is updated with the new tail pointer, which is the tail of the FID.

When the free buffer manager generates the discard command, then the following steps are taken:

- The cell is dropped and the BID is returned to the free buffer list. There is no other operation required.

When the timeout manager generates the discard command because of reassembly timeout, then the following steps are taken:

1. The FID memory is read. The previous packet tail is parsed.
2. The BID memory is read and the BID of that location is read. The content is the head of the new packet being enqueued.
3. The Free Buffer tail is read.
4. The BID memory location of address FBT will have the content the head of the packet that is being discarded. The buffers are on the free buffers list.
5. All the buffers are returned to the free buffer list.
6. The FID memory location is updated with the new tail pointer.

When the RED manager generates the discard command, then the following steps are taken:

1. The cell is dropped.

Port Management

On each SOP enqueue, the enqueue engine will check the port register to determine if the packet has to be dropped or not. The port number is available from the statistics memory. If the port register indicates that the cell has to be dropped, then the SOP cell is dropped. If the cell is not SOP, then the drop packet flag in the FID memory is checked. If the flag is set then the cell is dropped, else the enqueue operation continues. The port packet discard flag for that FID is set. The flag will be reset when the port register indicates that the port is re enabled again.

CLASS buffer Management

Enabling the buffer management bit in the control register will enable either the RED algorithm or the CLASS management. If the RED bit is reset, in addition to the previously enabled bit, then the CLASS management algorithm is enabled.

The enqueue engine will increment the counter in the CLASS register corresponding to that FID on every cell. The Dequeue engine will decrement the counter on dequeue command. When enqueue an SOP cell, the counter of the corresponding class.
On each SOP enqueue, the class counter will be compared to the threshold counter of that class. If the counter value exceeds the threshold, then the SOP cell is dropped. The cells of the rest of the packet are dropped until the next SOP cell. At that point the counter value is checked against the threshold and the process is repeated.

Random Early Discard Management

Enabling the buffer management bit in the control register will enable either the RED algorithm or the CLASS management. If the RED bit is set in addition to the previously enabled bit, then the RED algorithm is enabled. Another way to bypass the RED is by making both thresholds equal and very high. The average queue will not get as high as the threshold. The drawback is the calculations are still needed. The queue size and the averages are calculated according to cell size. The Queue size is updated every time there is an EOP. It is incremented by the number of cells. The queue size is decremented every time there is a dequeue operation. It is decremented by one.
When an SOP of an FID is received, the RED algorithm decides if the packet gets discarded. If the packet is discarded, then the SOP cell is dropped as well as subsequent arriving cells of the packet are dropped until the EOP cell is received. The BID is returned to the Free Buffer List.

With constant average queue size the goal is to discard packets at regular intervals in order to avoid clustering of discards and global synchronization. It is not desirable to have too many discarded packets close together and also it is undesirable to have a long interval between discarded packets.

The RED algorithm calculates the queue average size when an SOP cell is received and compares it to two thresholds. If the size is less than the minimum threshold, then the packet is not dropped. If the queue is larger than the maximum threshold then the packet is dropped. If the average size is between the minimum and the maximum thresholds, then a discard probability is calculated to determine the fate of the packet. There is a random number that will be used to determine if the packet has to be discarded in this case.

The first step when a packet is received by an empty queue is to estimate the number of packets that would have been transmitted during idle time. Then the average queue size will be calculated using the original estimate as if the number of packets has been transmitted during that interval. The average queue size is calculated for every FID. There is 1 K parameter memory for the various attributes such as the minimum and the maximum thresholds and the weight of the queue. There is an association between the FIDS and the attributes in the parameter memory.

The MAX_{th} is decreased when the probability P_a of discard is increasing. When the probability changes slowly the queue average size changes slowly. This will discourage oscillation in the average queue size. The values of attributes MINth and MAXth determines the utilization of the bandwidth

RED algorithm needs the following fixed and calculated parameters.
The following variables are calculated and saved:

- "AVG": The average queue size.

- "$Q\text{-}^{TIME}$": The start of queue idle time.
- "COUNT": The number of not discarded packets that have arrived since last discard.
- "Q": It is the current queue size. It is incremented on enqueue with the number of cells and decremented with one on dequeue.

The following parameters are fixed and programmable:

- "W_q": It is the queue weight. An initial value of 0.002 is used. It should not be large in order to filter transient congestions.
- "MIN_{th}": It is the minimum threshold for queue. If the traffic is bur sty, then the value has to be fairly large.
- "MAX_{th}": It is the maximum threshold for queue. The value determines the maximum average delay in the device. An initial value of 3 * MIN_{th} is used.
- "MAX_p": It is the maximum value for a probability "P_b" of discard. An initial value of 1/50 is used. It should never be greater than 0.1.
- "S": Typical transmission time for a small packet.
- "RANDOM": Random number from generator.

The following variables are calculated but not saved:

- "P_a": It is the current packet discard probability.
- "TIME": It is the current time.
- "F (*t*)": It is a linear function of time *t*.
- "M": The number of packets that would have been transmitted when queue is empty.

The following calculations are needed:

- The number of packets "m" that might have been transmitted during idle time.
- The average size of the queue.
- The probability of discard when the queue size is between the minimum and the maximum thresholds.

Equations

The queue empty size "M" equation is the following:

$$M = (TIME - Q\text{-}^{TIME})/S$$

The packet discard probability P_b equation is the following:

$$P_b = MAX_p\ (AVG - MIN_{th})/(MAX_{th} - MIN_{th})$$

It is desirable to have P_b a negative power of 2.

The final packet discard probability P_a equation is the following:

$$P_a = P_b / (1 - COUNT * P_b)$$

The average queue size is calculated according to the following:

$$AVG = (1 - W_q) * AVG + W_q * Q$$

Implementation

It is an easy determination when the average queue size is larger than MAX_{th} and is smaller than MIN_{th}. The most complex part is when the average queue size falls between the MIN_{th} and the MAX_{th}. When the queue average size is less than MIN_{th}, then the packet is transmitted. On the other hand when the queue average size is more than MAX_{th} then the packet is discarded. If the queue average falls between the MIN_{th} and

MAX_{th} then the probability determines the packet destiny. A pseudo random number "R" is generated by a uniform distribution random numbers generated.

The packet is discarded if "R" satisfies the following equation is satisfied:

$R < P_b / (1 - COUNT * P_b)$

If the average queue remains constant then the random number "R" is selected from the uniform distribution. The N_{th} packet is discarded if the following equation is satisfied:

$R \leq P_b N$

Random Selection

The random number is generated using linear feedback shift registers. The Ethernet polynomial will be used to generate a random number. The number is generated every time there is an enqueue operation. The following polynomial is used to generate the random number.

$G(X) = X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X^1 + 1;$

The value of the shift register is used as the number generated. The number of nibbles that is stored in the attribute table is used to determine the random number is used. If the value stored is "FF", then all eight nibbles are used. If the value is "7F", then the first 7 nibbles are used. See Figure 284.

Memory Size & Total Number of BIDS

See Figure 285.

BID Memories Chip Select

See Figure 286.

EXTRA MEMORY CYCLES

The extra needed cycles of the FID memory are the following:

- If the queue becomes empty after a dequeue operation the tail in the enqueue memory has to be written as "NULL".
- If the queue is empty before an enqueue operation the head in the DEQUEUE memory has to be written with the same value as the tail pointer.
- The Dequeue operation has to decrement the Qsize in the statistics memory.

The BID memory requires an extra cycle after a tail is linked in memory. The extra cycle will link the tail to a "NULL". When the head reaches a "NULL", then the queue is empty.

Memories Byte Enables

The state machines have to drive the byte enables to the various memories. There are some write cycles during which only selected byte enables have to be asserted. The rest of the byte enables should not be asserted.

The assertion of selected byte enables during write operations eliminates the need for a read operation.

- Read Operations for all memories: The state machines will assert all the byte enables during all read operations.
- Write Operations:

- Enqueue Memory: When the queue becomes empty after a dequeue operation, the tail pointer in the enqueue memory for that FID has to be written as NULL. When writing that location, the byte enable to bits 27 downto 0 will only be asserted.
- Dequeue Memory: When the queue is empty and the there is an enqueue operation, the head pointer in the de-queue memory for that FID has to be updated it. When writing that location, the byte enable to bits 36 downto 0 will only be asserted because the TYPE comes with the FID.
- BID Memory: When the NULL pointer is written at the end of an enqueue operation, the BID memory location has no relevant data. All byte enables will be asserted.
- Statistics Memory: The dequeue engine has to decrement the Qsize of that FID after a dequeue operation. The STAT memory is read to get the value. Therefore, the all byte enables are asserted.

Global Synch and internal Time slot generation

The global synch is provided for all the blocks. The block has to generate locally eight time slots. The memories controllers access these memories only during certain time slots. It is important that the all tasks are completed in one cycle of eight time slots or clocks. The tasks can take longer than eight cycles to complete. But the caveat is that a task has to start every eight cycles or time slots. The latency is longer but the bandwidth is the same. The scenario complicates the data dependency of all the concurrent tasks and sequential tasks.

External and Internal Memory Initialization

On power up the memories are not initialized. There is an initialize signal that when is set by the PCI, it triggers state machine that will write a zero pattern in all memory locations of all memories except the BID memory portion of the free buffer lists. In the free buffer lists, the locations will be initialized to a value that reflects the previous BIDS. The first location of the free buffer list will have the value X"00000000". The second location will have the value X"00000001". The rest of the locations will have an incrementing values. These will be the free buffer values that are used for BIDS. The first BID "000000" is not used. It is the "NULL" pointer.

Data Dependencies and Coherency

The data and the FID of both the enqueue and dequeue previous operation that has just finished have to be saved. The current FID has to be checked against the previous ones. If the current FID is the same as the previous one of either the enqueue and dequeue operations, then the data that was saved will be used. The data in memory is not the correct one.

There are several scenarios where the data coherency will get compromised. The data coherency will get compromised with consecutive enqueue operations to same FID:

- When the consecutive enqueue operations are for the same FID the statistics counters should be updated form internal registers.

- When the consecutive enqueue operations are for the same FID the tail pointer should be updated from internal register.
- When the consecutive enqueue operations are for the same FID the RED Calculations should be based on data in internal registers.
- When the consecutive enqueue operations are for the same FID the Q Size should be incremented based on data in internal register, then written in the statistics memory.

The data coherency will get compromised with consecutive dequeue operations to same FID:

- When the consecutive dequeue operations are for the same FID the head pointer should be updated from internal register.
- When the consecutive dequeue operations are for the same FID the Q Size should be decremented based on data in internal register then written in the statistics memory.

The data coherency will get compromised with enqueue and dequeue operations to the same FID:

- The only time that is compromised when there is an enqueue and a dequeue operation to the same FID. If the dequeue operation renders the queue empty and a "NULL" has to be written as the tail in enqueue memory and the head in the dequeue memory. Instead with the enqueue operation, the queue will not be empty any longer. The enqueue operation will preempt the dequeue operation and the new BID will be the tail and the head pointer.
- The Qsize in the statistics memory has to be update correctly. The number of cells is decremented by one before it is added to the Qsize.

The data coherency will get compromised with enqueue and timeout operations to the same FID:

- The data written to the FID memory has to be the enqueue one. The TTL increment and the discard operation have to be bypassed. There will be no discards of the queue. The TTL will be written as "00".

See Figure 287.

Statistics

The statistics memory is accessed when there is an enqueue, dequeue and CPU operations. All the requests are issued simultaneously.

The enqueue will update the various counters depending on the command issued by the memory manager.

- Enqueue command will increment by the number of cells the Q Size counter on EOP.
- Enqueue command will increment the packet counter on EOP.
- Enqueue command will increment the byte size counter on command.
- Discard command will increment the others discard counter.

The dequeue command will decrement by one the Q Size counter always.

The RED will increment the RED discard counter if the cell is dropped on EOP.

NULL Pointer

The value of "7FFFFF" is the "Null" pointer when the BID memories are all populated.

Shaper and Scheduler Commands flow

When it receives an EOP of an FID, then the PFQ will send a shape command to the shaper with the cell count of that packet. The Cell or packet indication has to have the correct value.

In case of ATM cells, the PFQ will send to the shaper a command to shape the FID cell.

External Interface

The external interface will be a standardized interface complying with the global SYNC. There are four interfaces between the PFQ and the external world. External memories are part of the PFQ and the interface is considered internally for all practical purposes.

- Enqueue a FID. The memory manger drives the request. The PFQ will respond with a BID and a valid signal.
  - The memory manager to PFQ command is issued in CLK 6.
  - The PFQ to memory manger response is issued in CLK 6, next synch.
- Discard a FID.
  - The memory manager to PFQ command is issued in CLK 6.
- Release BID.
  - The memory manager to PFQ command is issued in CLK 6.
- The second interface between the PFQ and memory manager is to dequeue an FID. The PFQ drives the dequeue request to the memory manager.
  - The PFQ to memory manager command is issued in CLK 6.
- There is an interface between the PFQ and the shaper. The PFQ drives the request to the shaper.
  - The PFQ to shaper commands are issued in CLK 6.
- There is an interface between the PFQ and the scheduler. The scheduler drives the requests to the PFQ. The request is to dequeue a BID of an FID.
  - The scheduler to PFQ command is issued in CLK 7.

Memory Structure

RED Attribute Memory

This is internal memory. There are 1K entries of this memory. The software writes it only. The address is the "RED_ASSOCIATION" from the FID memory. See Figures 288 and 289.

Statistics Memory

This is an external memory. There are 2 MEG locations. There are 2 locations per FID. The software writes only the port number. The rest is written by the logic. The CPU reads the counter values. The address is the "FID" for bits 20 downto 1with bit 0 as "0" or "1".

See Figures 290, 291 and 292..

All discard counters are updated on EOP. If the packet is being dropped because of RED and the EOP cell has a discard command instead of enqueue, then the RED counter is not update it.

FID Dequeue Memory

This is an external memory. There are 1 Meg locations. There is one location per FID. The software does not write any location. The software under normal operating traffic conditions does not need any values from this memory. The address is the "FID". See Figures 293 and 294.

FID Enqueue Memory

This is an external memory. There are 2 MEG locations. There are 2 locations per FID. The software writes only the RED association. The software under normal operating traffic conditions does not need any values from this memory. The address is the "FID" for bits 20 downto 1with bit 0 as "0" or "1". See Figures 295, 296 and 297.

Buffer ID Data Structure

This is an external memory. There are 8 MEG locations. There is one location per BID. The software writes the count only. The count cannot change when the queue is not empty. Hence there is no need for a separate byte write enable. The address is the "BID" tail or head that are read from the dequeue memory, enqueue memory or the free buffer register. See Figures 298 and 299.

The SOP, EFCI, OAM are set when the SOP cell arrives. The EOP is set in a different BID when it arrives. There is no need to write extra cycles on EOP.

Multicasting Data Structure

These are internal memories.

Multicasting FID LEAF Data Structure

Multicasting FID Dequeue Memory

There are 1 K locations. There is one location per FID. See Figures 300 and 301.

Multicasting FID Enqueue Memory

There are 2 K locations. There are two locations per FID. There are 2 locations per FID. The software writes only the RED, and "NEXT_TUN". The software under normal operating traffic conditions does not need any values from this memory. See Figures 302, 303 and 304.

Multicasting FID ROOT Data Structure

The enqueue engine uses the next table. There are 1 K locations. The software writes the head and the tail. See Figure 305.

Tunneling Data Structure

The memories are internal.

Tunneling FID LEAF Data Structure

Tunneling FID Dequeue Memory

There are 1 K locations. There is one location per FID. There is one location per FID. The software does not write any location except the "TUN_ROOT". The software under normal operating traffic conditions does not need any values from this memory. See Figure 306.

Tunneling FID Enqueue Memory

There are 2 K locations. There are 2 locations per FID. There are 2 locations per FID. The software writes only the RED, and "NEXT_TUN". The software under normal operating traffic conditions does not need any values from this memory. See Figures 307, 308 and 309.

Tunneling FID ROOT Data Structure

The de-queue engine uses the next table. There are 1 K locations. The software writes the head and the tail. See Figure 310.

Interface

The interface is either internally to the memory manager, shaper or the scheduler. Else, it is to the external rams. The BYTE enable for the enqueue memory should be connected that the "BYTE_ENABLE_0" is connected to the first three bytes that contain the BID tail pointer. The second "BYTE_ENABLE_0" should be connected to the rest of bytes. When writing the BID tail pointer with the value "NULL" by the dequeue engine, there is no need for a read operation, instead only the first three bytes are written. The rest of the data field is maintained as it was. The dequeue memory has no relevant data that need to be maintained when the enqueue engine updates the BID head pointer after empty queue. The type field is updated with the pointer. The "TYPE" is part of the

command. See Figure 311. The signal "MEM_PFQ_FULL" is synchronized and bussed through to the DBS.

CPU Interface

There are signals that are generated by the CPU interface and are used to control the memory sizes and the enable and disable signals. See Figure 312.

Segmentation Interface

See Figure 313. The PFQ of an egress device sends a message to the segmentation engine of Ingress devices when the number of used buffers exceeds a programmed threshold. The data will contain the port address of the egress device. The segmentation engine will multicast the message to all the input devices that instructs these devices to refrain from sending any traffic to the device of the corresponding port. See Figure 314. The message indicates the condition when the number of buffers used exceeds the threshold value by setting the flag to '1'. There is another message that indicates when the condition no longer exists by setting the flag to "0". The rest of the data bits indicates the port number that is programmed by the CPU.

DBS & CBWFQ Interface

See Figure 315.

DBS Interface

See Figure 316.

Memory Manager Interface

A command is stretched to few cycles as described below so that the number of pins is reduced. The interface between the 2 blocks is shown in Figures 317 and 318. The En-queue/Discard command from the memory manager to the PFQ takes 2 cycles to complete. The De-queue command from the PFQ to the memory manager takes 3 cycles to complete. When the BID is not valid, the "NULL" BID will be returned to the memory manager as the BID value. The memory manager will drop the cell whenever the BID is "NULL". See Figures 319 and 320.

MEM to PFQ

The memory manager starts the command by asserting valid signal. The PFQ will always sample the command. If it is not NOP, then the command determines the number of valid cycles with data. If the command is a "discard" command, then the only valid parameters are the FID; all the rest of the parameters should be ignored by the PFQ. If the command is a release then the BID is important and the rest is ignored. If it is an enqueue command, then both D1 and D2 are valid. See Figure 321.

The memory manager as well as the PFQ should have FIFOS to store the commands and the parameters.

PFQ to MEM

The command takes 3 cycles to complete. In this command the data fields are transferred in 3 cycles as shown in Figure 322.

If "REL" is set, then the memory manger can reuse the buffer, which is the BID. As a result, the memory manger cannot send a command to PFQ releasing BID that was dequeued with the "REL" bit reset. The PFQ will not check the "DEQ CNT" in the BID memory. The "DEQ CNT" is used for multicasting traffic multiple times.

Interface Timing between the Memory Manger and PFQ

The MX1 and MX2 are not synchronous. The global synch signal is generated locally. As a result, the time slot is not adhered to in this interface. All the commands or data are stored in FIFOS after they reach the receiving device. The response to such a request will take between 16 and 24 clock cycles. For instance, the memory manger needs to wait between 16 and 24 clocks before it gets a BID after it asserts the FID enqueue command. See Figure 323.

Memory Manager Interface Timing

See Figure 324. The clock is 100 MHZ and the commands from the PFQ to the memory manager are valid every 4 clocks. Similarly, the commands from the memory manager to the PFQ are valid every 4 clocks. As a result, the PFQ block and the memory manager block will receive a command every 8 clocks of 200 MHZ.

SEGMENTATION Interface Timing

See Figure 325. The clock is 100 MHZ and the data is serially driven on two data pins. The flag indicates if the queue is full when it is "1". When the queue is full and the number of buffers used is equal or exceeds a programmable threshold, the PFQ will send a message indicating the condition by setting the flag to "1". When the number of buffers used is less than the programmed threshold by 10, then an equivalent message is sent but with the flag reset.

Software Interface

The CPU interface doesn't use bursting. Data bus is still 32 bits.

Features

- Data bus of 32 bits. There is data bus in and out.
- Single address access (not a burst access).
- CPU accesses all registers in all blocks via direct read/write commands.
- All external memories and most of the internal memories are accessed via indirect accesses through registers
- CPU accesses all internal/external memories in all blocks via indirect accesses using the general-purpose registers dedicated for these accesses.

Address space

The address space is 64 entries only. The internal registers directly mapped to these addresses and external/internal memories are indirectly mapped.
The address spaces per lookup engine is shown in Figure 326.

Direct/Indirect Accesses

A direct access is used to access. If there are internal memories or registers, which are longer then 32 bits, then read or write to these locations will be executed in more then one access. For example an entry in internal memory of 50 bits width (or a register with the same length) will be accessed in two CPU commands first access bits [31:0] and then [49:32]. Each register or small internal memories have their own address, which is unique to the entire chip set.

An indirect access is used to access internal and external memories. Each block has a set of registers dedicated for the indirect access. One register contains the address and the command (Rd, Wr, Init etc). Few other registers contain the data for a write command. Writing to the COM/ADD register triggers the execution of the command, once the block completed the execution of the command, the COM portion of the command is cleared by the block. When a read command occurs the data is written to the registers and then the CPU reads the registers to get the data.

Registers

See Figure 327.

Commands

NOP.

See Figure 328.

Read FID Enqueue Memory (72 bits) – Up to 2M locations.

See Figure 329.

Write FID Enqueue Memory (72 bits) – Up to 2M locations.

See Figure 330.

Read FID Dequeue Memory (36 bits)– Up to 1M locations.

See Figure 331.

Write FID Dequeue Memory (36 bits) – Up to 1M locations.

See Figure 332.

Read BID Memory (36 bits) – Up to 16M locations.

See Figure 333.

Write BID Memory (36 bits) – Up to 16M locations.

See Figure 334.

Read Stat Memory (72 bits) – Up to 2M locations.

See Figure 335.

Write Stat Memory (72 bits) – Up to 2M locations.

See Figure 336.

Setup connection command for FID

See Figure 337. This command sets up a connection by accessing external memories and initializes the corresponding locations to the proper parameters. It is a write command. Clear PFQ entries for ENQ, DEQ and STAT!

Tear-Down connection command for FID

See Figure 338. This command tears down a connection by accessing external memories. It is a write command. The data is all "0"

PFQ returns all remaining BIDS on the queue of the down connection to the free link list during this command.

INIT FID MEMORIES

The memories are the enqueue, dequeue, and statistics memories. See Figure 339.

INIT BID Memory

This will build the free buffer list. See Figure 340.

Reserved Op-codes

The op-codes 1101, 1110, and 1111 are reserved.

General commands

The following commands are combinations of the above simple commands. A user command like this should be broken to several Read or Write commands in the driver.

Initialized External memories

- Write "0" in all memory locations.
- Write "NULL" in the enqueue memory.
- Write "NULL" in the dequeue memory.
- Build the free buffer list by linking each buffer to the next one.

Initialized Internal memories

- Write "NULL" in the Multicast Root memory.
- Write "NULL" in the Multicast Leaf memory.
- Write "NULL" in the Tunneling Root memory.
- Write "NULL" in the Tunneling Leaf memory.

Principle Of Operation

The PFQ performs the following operations:

- PFQ adds a BID to the FIDS queues. In essence it enqueues a BID.
- PFQ removes a BID from the FIDS queues. In essence it dequeues a BID.
- PFQ adds a BID to the free buffer queue. In essence it de allocates or returns a free BID.
- PFQ removes a BID from the free buffer queue. In essence it dequeues or allocates a free BID.
- PFQ increments the statistics counters.
- PFQ timeouts the active FIDS.
- PFQ asserts commands to the shaper and memory manager.
- PFQ accepts commands from the scheduler and memory manager.
- PFQ transfers statistics to the CPU.
- PFQ executes early random discards depending on criteria that associate the FID with a counter and an algorithm.
- PFQ executes discard operations.
- The Memory manager issues the enqueue, discard and the release.
- The scheduler issues the dequeue command.

The Per flow queue will get a command from the memory manager to enqueue an FID. The PFQ rejects or honors the request depending on the free buffers available or RED. The request is honored and the enqueue operation is performed. If the request is rejected, then a discard operation is performed by making the tail of the discarded packet

as the tail of the free buffer list. Also, the head of the packet will be linked to the previous free buffer tail.

The scheduler commands are dequeue ones. The PFQ will remove the BID from the queue and transfer the information to the memory manager.

Error Conditions

The total numbers of free buffers is compared to the number of buffers used and also to the total number of buffers on the queues. If the numbers do not match, then there will be an abort of all the operations. The PFQ will indicate the condition to the CPU. Another error condition happens when there is a dequeue request and the queue is empty.

TIMING DIAGRAMS

The timing diagrams are for FID and BID memory accesses. They include the accesses of the FID memory as well as the BID memory for enqueueing, dequeueing, statistics collections and timeout scanning. See Figure 341.

The memories cycles are summarized in Figures 342-346.

When dequeues engine is updating the tail pointer in the enqueue memory to "NULL", there is one write operation that is performed to the enqueue memory location 0. The BYTE ENABLE for the first 24 bits should be used. All the other byte enables should not be asserted.

When the enqueue engine is updating the head pointer in the enqueue memory to "BID", there is one write operation that is performed to the dequeue memory. The BYTE ENABLE for the first 24 bits should be used. All the other byte enables should not be asserted.

Per Flow Commands

- The Memory manager command to PFQ:
  - ENQ {FID}: Enqueue the FLOW.
  - DSCD {FID}: Discard the FID.
  - REL {BID}: It releases the BID.
- The PFQ Commands to the memory manager:
  - SND {BID}{REL}{PORT}: Get the data cell belonging to BID and send it with port number "PORT", afterwards release the buffer if "REL" is set.
  - VALID {BID}: Accept the BID for the previous enqueue command.
- The PFQ Commands to DBS:
  - SHAPE_OR_SCHEDULE {FID}{BID}{LOCATION}{#CELLS}{EOP}: Shape the current Flow ID FID with the location "LOCATION", EOP marker received even though should be implied.
- The DBS Commands to PFQ:
  - DEQ {FID}: The PFQ will pass the Buffer ID to the memory manager. If EOP is set, then send a shape command to shaper.
- The PCI Commands to PFQ:
  - Get [ADR]{CNTS}: Get the counts values from the statistics memory.

Data Base

The data base block is responsible for managing the packets link list per FlowID for FlowIDs in the Shaper and FlowIDs in the scheduler. The block receives an En-queue command from the PFQ and sends the FlowID either to the Shaper or to the Scheduler. A De-queue command starts with a port selection based on a port calendar and then a FlowID is selected between the scheduler and the shaper.

Shaper traffic has priority over scheduler traffic. Shaper traffic is divided to 8 classes of strict priority, once all shaper's classes are empty for a given port, the scheduler is allowed to schedule.

Features

1. This block operates at 200MHz clock.
2. Support up to 1M FlowIDs.
3. Support a link list of packets/cells up to 8M cells.
4. Performs the following in one port slot (8 cycles):
    a. One input phase from the PFQ and redirect to an input phase in the Shaper or in the Scheduler or to both (when traffic is just metered)
    b. One output phase of the Shaper.
    c. One output phase of the Scheduler.
5. Support 8 strict priority classes for the shaped traffic per port.
6. Performs port scheduling for the shaped traffic base of strict priority within the block.

Interface

A global block diagram of the database block is shown in Figure 347. Figure 348 shows the interface to/from the data base block. See also Figure 349.

Functional Description

The DBS block acts as the holder of the information per FlowID and the packets within the FlowID for the Shaper and the Scheduler blocks. The DBS block can be divided to several processes during one port slot:

PFQ Input Phase

For a packet traffic the PFQ first assembles a complete packet and then transfer the packet to the DBS block, when in a cells traffic, a cell is being sent to the DBS each time it arrives. One cell in cells traffic is considered as a packet with a length of one cell. When a packet arrives to the DBS block from the PFQ, the DBS first links the packet to the FlowID and also generates an input phase to the Shaper or to the Scheduler or to both based on a parameter bit per FlowID.

Shaper Output Phase

The output phase of the Shaper is being performed on a cell basis, means each cell of a packet generates an output phase from the Shaper. Once a cell is driven from the Shaper, the DBS block decrement the cell counter for the specific packet. If it's the last cell of the packet, the DBS links the packet to the shaper's output memory per port. The

DBS also generates an empty indication to the Shaper if there are no more packets to shape for the FlowID.

Port Calendar

A port calendar is a free running port selection based on a memory programmed by the user. Once a port is selected for the current output phase, the port number is transferred to both scheduler and shaper's output memory. Both blocks select their output FlowID for the output stage. If the shaper's output memory has a valid FlowID and the scheduler is not in the middle of a packet, then the FlowID to be driven to the PFQ is always from the shaper's output memory. Shaped traffic has priority over un-shaped traffic.

Each output phase that starts with the port calendar also starts with checking the port status. Only if the port calendar has a valid port and that port is not full according to the port status, a valid port is driven to the scheduler and to the shaper.

Port Status

This block receives the status information from the output SPI interface for 64 ports plus one CPU port. See the timing diagram of that interface below.

Scheduler Output Phase

Once a port is selected, the scheduler performs it's own output phase. The DBS block needs to support this phase the same way it supports the shaper's output phase.

Block Description

The following are the algorithms to perform in each of the previous phases:

PFQ Output Phase

1. *The PFQ drives the following signals: VALID, FID, NCELL, BID, CLPI*
2. ***Read FID_mem[FID]*** *{ rp, wp, e, cell_cnt, ncell, clp, port, qos, c_p, CPU_port, shape}*
3. *n_wp <= BID*
4. *n_e <= 0*
5. *if(e==1)*
    1) *if (shape_meter==0 | shape==1) perform input phase to Shaper: VALID, FID, qos, ncell*
    2) *if (shape_meter==0 | shape==0) Perform input phase to Scheduler: VALID, FID, qos, CPU_port*
    3) *n_rp <= BID*
    4) *n_ncell <= NCELL*
    5) *n_cell_cnt <= NCELL*
    6) *n_clp <= CLPI*
    7) *pkt_next_we = 0*
6. *else*
    1) *if (shape_meter==0) perform input phase to Shaper: VALID, FID, qos, ncell*
    2) *n_rp <= rp*

3) *n_ncell <= ncell*
4) *n_cell_cnt <=cell_cnt*
5) *n_clp <= clp*
6) *pkt_next_we <= 1*

7. *if(VALID==1)* ***Write FID_mem[FID]*** *<=*
   {n_rp, n_wp, n_e, n_ncell, n_cell_cnt, n_clp, qos, c_p, CPU_port, shape}
8. *if(VALID==1 & pkt_next_we==1)* ***Write pkt_mem[wp]*** *<=*
   {ncell, clp, FID, next, [dbs_dpkt_word_en[1:0] =11] <= {NCELL, CLPI, 0, BID}

Shaper Output Phase

1. *The SHP drives the following signals: VALID, FID, VIOLATOR*
2. ***Read FID_mem[FID]*** *{rp, wp, e, cell_cnt, n_cell, clp, port, qos, c_p, CPU_port, shape}*
3. *Wait until read is complete*
4. *cur_eop <= cell_cnt==1*
5. *cur_empty <= cur_eop & rp==wp*
6. ***Read pkt_mem[rp]*** *{ncell, clp, next, FID}*
7. ***Read shp_out_port[port]*** *{p_rp, p_wp, p_e}*
8. *if(cur_eop==1) n_rp<= next; n_cell_cnt <= ncell; n_n_cell <= ncell;*
9. *else n_rp<= rp; n_cell_cnt <= cell_cnt –1; n_n_cell <= n_cell;*
10. *if(cur_empty) n_e <= 1*
11. *n_FID <= FID*
12. *n_clp <= en_clp_change ? (VIOLATOR | clp) : clp*
13. *if(cur_eop)*
    shp_out_port_we <= 1
    n_p_e <= 0
    n_p_wp <= rp
    if(p_e) n_p_rp <= rp
14. *else*
    shp_out_port_we <= 0
15. *send cur_empty indication to Shaper*
16. *if(VALID==1)* ***Write FID_mem[FID]*** *<=*
    {n_rp, wp, n_e, n_cell_cnt, n_n_cell, qos, c_p, CPU_port, shape}
17. *if(VALID==1 & cur_eop)*
    Write shp_out_port[port] *<= {n_p_rp, n_p_wp, n_p_e }*
18. *if(VALID==1 & cur_eop & p_e==0)*
    Write pkt_mem[p_wp][next][dbs_dpkt_work_en[1] =1] *<= {rp}*
19. *if(VALID==1 & cur_eop)*
    Write pkt_mem[rp][dbs_dpkt_work_en[0] =1] *<= { n_clp, n_cell, n_FID}*

DBS Output Phase (Combined with Scheduler Output Phase)

1. ***Read port_calendar mem[count]*** *{portid, valid, jump}*
2. *if(jump==1) count <= 0*
3. *else count = count + 1*
4. *send shaper and scheduler: portid, valid*

5. *wait for 4 cycles to receive SHP_FID, SHP_VALID, SCH_FID, SCH_VALID, SCH_MID*
6. *get PFQ_FULL indication from PFQ block*
7. *if(valid & ~PFQ_FULL & SHP_VALID & (~SCH_MID | ~SCH_VALID))*
    1) *out_from_shp <= 1*
    2) *out_from_sch <= 0*

   if(valid &~PFQ_FULL & SCH_VALID & (~SHP_VALID | SCH_MID)
    1) *out_from_shp <= 0*
    2) *out_from_sch <= 1*
8. *if(out_from_shp==1)*
    1) ***Read shp_priority_mem[portid]*** *{empty}*
    2) *Select class per portid based on empty : class_addr = {portid, class}*
    3) ***Read shp_out_mem[class_addr]*** *{p_rp, p_wr, p_e}*
    4) ***Read pkt_mem[p_rp]*** *{ncell, clp, next, FID}*
    5) *Send to PFQ: FID, portid, p_clp, valid(=1)*
    6) *If(ncell==1)*
        a. *n_p_rp <= next*
        b. *if(p_rp==p_wp) n_p_e <= 1*
        c. *n_ncell <= 0*
    7) *else*
        a. *n_p_rp <= p_rp*
        b. *n_p_e <= p_e*
        c. *n_ncell <= ncell – 1*
    8) ***Write shp_out_mem[class_addr]*** *<= {n_p_rp, p_wp, n_p_e}*
    9) *If(n_p_e==1)* ***Write shp_priority[portid]*** *<= new empty indication*
    10) ***Write pkt_mem[p_rp]*** *<= {n_ncell}*
9. *if(out_from_sch==1)*
    1) ***Read FID_mem[SCH_FID]***

{rp, wp, e, cell_cnt, n_cell, port, qos, c_p, CPU_port, shape}
    2) *eop <= (cel_cnt==1)*
    3) *empty <= (cell_cnt==1 & rp==wp)*
    4) *busy <= PFQ_FULL*
    5) ***Read pkt_mem[rp]****{ncell, clp, next,FID}*
    6) *Send eop, busy, empty to scheduler*
    7) *if(eop) n_rp <= next; n_cell_cnt <= ncell; n_n_cell<= ncell;*
    8) *else n_cell_cnt <= cell_cnt-1;n_ n_cell <= n_cell; n_rp <= rp;*
    9) *if(empty) n_e <= 1*
    10) ***Write FID_mem[SCH_FID]*** *<=*

{n_rp, wp, n_e, n_cell_cnt, n_cell, port, qos, c_p, CPU_port, shape}

Port Status

This interface drives the status of the 64 ports in the output SPI and the CPU port. It is a continuous interface. Every 8 cycles of 200MHz the status indication of the 65 ports is updated to a 65 bits register. The first cycle of the eight is marked with a specific signal (sync signal). Figure 350 shows the interface.

The database block needs to synchronize the incoming status bits to the local 200MHz clock and then use the status bits. A synchronization FIFO is being used, but since the write and read frequencies are the same, there is no empty or full indications. Reads will always lag 3 cycles after the writes. To achieve that, the read pointer is always the value of the write pointer minus 3. Also the first 3 reads after reset should be ignored.

Full indication from PFQ

A full indication from the PFQ can be asserted any time. Once a full indication is asserted, the DBS first complete the current output phase and then freeze the increments of the port calendar. From that moment until this indication is de-asserted no output phase is performed from the scheduler or from the shaper's output FIFOs. The shaper's output phase to the DBS should still continue without any connection to the full indication.

CLP Bit

The CLP bit is modified in the Database and then is sent back to the PFQ during the output stage. Only FlowIDs that are shaped can have their CLP bit changed, FlowIDs to the scheduler have the CLP bit unchanged. The violation indication from the shaper should set the CLP bit to 1, if no violation indication arrives then the CLP bit is to remain the same as it was.

Memory Description

Chip-Select indication for the PKT Memory

The packet memory can support up to 8M locations (the address bus has 23 bits). Since there are no SSRAM memories in the market that have that amount of entries, the DBS provides chip-select signals so that several memories can be connected to get up to 8M locations.

The chip-select signals are used only when supporting 2M locations and up. The generation of the chip-select depends on the input from the CPU interface called CPU_CS_SEL[1:0] as follow:

00 – generate chip-select based on address[20:19]
01 – generate chip-select based on address[21:20]
10 – generate chip-select based on address[22:21]
11 – Reserved

for example, if supporting 2M locations only and the memory devices used have a depth of 512K locations, then CPU_CS_SEL bus should have the value of 00. the chip_select[3:0] will have the following meaning:

chip_select_l[0] – selects memory block for addresses 0 – 1/2M
chip_select_l[1] – selects memory block for addresses 1/2M – 1M
chip_select_l[2] – selects memory block for addresses 1M – 1 1/2M
chip_select_l[3] – selects memory block for addresses 1 1/2M – 2M

In this case address bus bits [18:0] are going to all memory blocks (Note, each block has only 512K location, which translated to 19 bits), the chip_select_l bus controls 4 memory blocks, therefore, the total locations supported is 2M.

DBS FlowID Memory

FlowID Memory

Address: FID, Length: 1M, #bits: 91

See Figure 351.The user has to initialize the above fields, which are marked as software using setup connection command.

PKT Memory

Address: BID, length: 8M, #bits: 55

See Figure 352.

Port calendar memory

Address: counter, length: 192, #bits: 8

See Figure 353. The user should initiate this memory prior to any traffic based on the users input.

Shp_strict memory

Address: port, length:64, #bits: 12

See Figure 354.

Shp_out_port memory

Address: port, length:512, #bits: 49

See Figure 355

.

Registers Description

See Figure 356.

CPU Commands

Read external FID memory

See Figure 357.

A[0] – C_P (bit [6] in the register)
A[1] – CPU_PORT (bit [7] in the register)
Pri[2:0] – Priority for shaped traffic
S – Shape (bit [23] in the register)
C[0] – clpi (bit [10] in the register)
C[1] – Empty (bit [11] in the register)

Write external FID memory

See Figure 358.
A[0] – C_P (bit [6] in the register)
A[1] – CPU_PORT (bit [6] in the register)
Pri[2:0] – Priority for shaped traffic
S – Shape (bit [23] in the register)
C[0] – clpi (bit [10] in the register)
C[1] – Empty (bit [11] in the register)

Setup FID connection

See Figure 359.
A[0] – C_P (bit [6] in the register)
A[1] – CPU_PORT (bit [6] in the register)
Pri[2:0] – Priority for shaped traffic
S – Shape (bit [23] in the register)
During the setup connection command, the user defines the above fields only.
The fields RP, WP, NCELL, CELL_CNT and CLP should be reset to 0 by the hardware
The field EMPTY should be set to 1 by the hardware.

Read external PKT memory

See Figure 360.
L[0] – CLP
L[1] - Reserved

Write external PKT memory

See Figure 361.
L[0] – CLP

L[1] - Reserved

Read port calendar memory

See Figure 362.
Data[5:0] – portid
Data[6] – valid
Data[7] – jump

Write port calendar memory

See Figure 363.
Data[5:0] – portid
Data[6] – valid
Data[7] – jump

Read SHP Strict memory

See Figure 364.
P[2:0] – previous QOS pointer
P[3] – previous QOS pointer valid

Write SHP Strict memory

See Figure 365.

Read SHP out memory

See Figure 366.

Write SHP out memory

See Figure 367.

Init SHP_out_port memory

See Figure 368. This command generates write accesses to all entries in the Shp_out memory (64 entries) with a specific data of: p_rp = 0, p_wp = 0, p_e = 1

Init Shp_strict memory

See Figure 369. This command generates write accesses to all entries in the Shp_strict memory (64 entries) with a specific data of: empty = 8'b1111_1111, prev_qos_v = 0, prev_qos = 0

Reserved Opcodes

The opcodes: 1110, 1111 are reserved.

Timing Diagram

Output phase to PFQ from SHP output memory. See Figure 370.

Notes for Figure 370:

PFQ Output, SHP Output, SCH Output *DBS Output*

Notes :

1.Slot 4 has data dependency on slot 6 if SHP FID and PFQ FID are same. Use following algorithm for this data dependency

```
if (shp_FID == PFQ_FID & shp_out_valid & PFQ_in_valid(
@slot 6   empty <= 0
          wp <= BID(PFQ)
          if (shp_empty) rp <= BID(PFQ)
          else rp <= n_rp
```

2. Slot 4 has data dependency on slot 7 if scheduler FID is same PFQ FID. In that case

```
if (sch_FID == PFQ_FID && sch_out_valid & PFQ_in_valid)
 @slot 7
          empty <= 0;
          wp <= BID(PFQ)
          if (sch_empty) rp <= BID(PFQ)
          else rp <= n_rp;
```

There are no data dependencies on the pkt memory since a packet can exist only in one place at a given time.

Interfaces

Shaper

See Figure 371.

Scheduler

See Figure 372.

PFQ

See Figure 373.

Output Scheduler

The output scheduler receives unshaped flow IDs from the Data Base Module(it is referred as DBS) and links them in its database. Then it chooses a flow ID based on its scheduling algorithm and outputs the flow ID to the PFQ engine. Note that unshaped flow IDs will get available bandwidth which is not used by shaped FLOW IDs. Shaped flow IDs are scheduled for output by the DBS and these flow IDs get committed bandwidth. In other words Shaped flow IDs from the DBS always get priority over unshaped flow IDs from scheduler unless it is middle of packet. When DBS received a flow ID, flow ID is either sent to shaper(if it to be shaped) or scheduler(if it is not to be shaped). Once flow ID is received from DBS, scheduler links this flow ID based on the port number and QOS. Ports are always serviced in round robin fashion by the database. DBS informs the scheduler the port number which will be served in this time slot(8 clocks). If there are one or more flow IDs pending for this port then a flow ID is selected for scheduling based on the following criteria.

The scheduler implements 2 levels of scheduling

- Level 2: Weighted-Round-Robin algorithm combined with strict priorities and best effort to schedule QOSs.
- Level 3: Round Robin between FlowIDs

The scheduler also interfaces with a CPU through the software interface. The CPU can read or write the scheduler internal registers and all the memories. Furthermore, the scheduler receives flow control status from the re-assembly engine, the PFQ engine and the egress chip.

Features

1. The scheduler operates at 200MHz clock.
2. Round-robin algorithm between Flows.
3. Combination of weighted round-robin algorithm and strict priority for QOSs.
4. Support up to 1048576 Flows (1M).
5. Support 192 port of STS1 rate each up to one OC192 port. CPU port is always OC3.

6. Supports up to 1024 QOSs that are shared between the output ports.
   - Up to 8 QOSs per port. CPU port has only one QOS, which is not accounted in the 1K.
   - The 8th QOS per each port is acting as a best effort QOS, means this QOS will be selected only if all other QOSs are empty.
7. Two modes of operations:
   - Egress Mode: (Normal operation) Each port has up to 8 QOSs, there are 64 output-ports plus 1 CPU port which has only one QOS.
   - Ingress Mode: Support only one output port. All QOSs belong to that one port. Virtually there are still up to 256 ports with up to 8 QOSs dedicated to each. A separate port status information bits will arrive from the Egress chip to block the QOSs in the Ingress chip, which send traffic to the congested ports in the Egress chip. This mode of operation affects the port calendar only.

   Figure 374 shows both applications with respect to the switch:

8. Status for Ingress chip applications comes from the Egress chip.
9. External memory supported is pipelined ZBT memory. Micron provides pipelined ZBT memories of 512K x 32/36 or 1M x 18.
10. In case of scheduling packets, only one packet can exist in the scheduler for a specific FlowID. Once this packet is scheduled the database block (DBS) will issue another packet of the same FlowID to the scheduler if there is one linked in the DBS. In case of scheduling ATM cells, each cell is treated as a one cell packet.
11. Accept to link a flow from the DBS once every 8 cycles of 200MHz.
12. Since 8 clock cycles are needed to service a port (a cell), a total of 520 (64 ports + CPU port) cycles are needed to keep up the 10Gbs traffic rate.

Interface

Figure 375 shows the interface to/from the scheduler block. Figure 376 shows the detailed interface to other blocks. Signal Descriptions are found in Figure 377.

Functional description

Flow ID is given to scheduler by DBS if flow ID given by PFQ is not to be shaped. Based on a port and a QOS a flow ID belongs to, the scheduler informs DBS to link up the flow ID in the DBS's linked list memory structure. DBS selects port to serve and inform scheduler port number which will be served in the given slot. Scheduler then selects a QOS based on the strict priorirty or WRR algorithm among all the QOSs of the port, and finally chooses a flow ID based on the round robin algorithm between all the flows of the QOS. The scheduler sends the scheduled flow ID to DBS for scheduling. At this point DBS decides whether flow ID from scheduler will be sent out or flow ID from shaper will be sent out. Always if there is flow ID from shaped traffic for this port, it will be sent out first. In that case DBS informs scheduler not to update any parameters as flow id from scheduler has not been scheduled. When there is no flow ID from shaped traffic for this port, then flow ID from scheduler will be sent out. In that case, scheduler updates all parameters. Also if the current cell scheduled is the last cell (in case of ATM traffic, Every cell sent out will be marked as eop) then End Of Packet(eop) signal is sent from DBS to scheduler. Also if the current packet is the last packet linked for this flow ID,

empty will be asserted by DBS to scheduler. If it eop and not empty then scheduler has to rotate flow IDs. Scheduler will link current read pointer(f_rp) to end of link(f_wp). This information is sent to DBS. Also next_FID given by DBS will become head of link list for this QOS.

The following explains all the scheduling algorithms employed by the scheduler and the scheduler's linked list memory structure.

Round Robin Algorithm

All the elements to be scheduled are linked together in one link list. There are two pointers to that link list, a head pointer and a tail pointer. A selection is basically forwarding the head pointer. There are 2 options to continue the algorithm after selecting the element: the element is linked to the end of the list and become the tail pointer while the next element becomes the head of the list. The second option is that the element is removed from the link list, in this case only the pointer is changed to the next element in the list. See Figure 378.

A Round Robin algorithm is used for the FlowIDs. The port calendar is implementing the same but without removing or adding elements to the link list, means a fixed list.

Weighted-Round-Robin Algorithm

This algorithm is an extension to the Round Robin one. Each element in the link list has a weight and will be selected as much as the weight allows. This algorithm should generate a relative bandwidth to each element according to the total elements linked:

$$BW_k = Weight_k / \sum Weight_i$$

To implement this algorithm there is a need for two link lists. Active link list, which holds the active elements to schedule, once an element k was schedule $weight_k$ times, it is moved to the waiting list. Once all the elements in the active list exhausted their weight and moved to the waiting list (active list becomes empty) the waiting list becomes the active list and the above formula is performed on the elements in the new active list. During processing of the active list, if a new element arrives, it is linked to the waiting list and will participate the scheduling only after the current cycle of the above formula is done. Each time an element is moved to the waiting list, the weight count will be set to the original weight. See the example in Figure 379. This scheduling scheme is used to schedule the QOSs per port.

Smoothing the bursts

The WRR algorithm can cause long bursts of cells for the same FlowID/QOS if the weight is high. In order to make smaller bursts of data for the same FlowID/QOS, the QOSs are rotating in the active list several times before exhausting their weight. A Factor parameter is introduced per port, the weight of all QOSs is divided by the Factor and the result is the number of times they will be scheduled before moving to the end of the active list. For an example see Figure 380.

Negative Weight

When scheduling packets, the QOS of the specific FlowID will be scheduled until an EOP indication arrives, it is possible that the packet is bigger then the weight of the

QOS so that the count can become negative. Once EOF arrives if the weight count is still positive, the QOS can be scheduled for another packet. If the weight count is negative, the QOS is linked to the waiting list. The weight of the QOS is added to the weight count as well. Once he waiting list becomes active and the QOS is to be scheduled, if the weight count is still negative weight will be added again and the QOS will be sent to the waiting list again without scheduling.

This is similar to a negative credit, if a QOS consumes all the credits and beyond it has to accumulate enough credits in order to start scheduling again.

Best effort QOS

This is another flavor for the selection of the QOS per port. A dedicated QOS (QOS7) is called the best effort QOS. It does not participate in the strict priority scheme or in the weighted-round-robin scheme. This QOS is scheduled only if all QOSs that belong to the strict priority are empty and all the QOSs that belong to the WRR are empty as well. A simple way to describe it is that this QOS is scheduled only if all other QOSs are empty.

Strict Priority

Select a QOS can be done either by WRR algorithm or by assigning priorities to QOSs. Each port can assign priorities and WRR QOSs as shown in Figure 381.

Selecting QOS

As shown in Figure 382 there will be two QOS queues maintained per port. One is active list and other is waiting list. When a flow ID arrives from DBS, it will always be put in waiting list. Since there are eight QOSs per port each of this list is 8-bit wide and they are called QA_EMPTY(active list) and QW_EMPTY(waiting list). When corresponding bit is 1 then this QOS is empty and nothing to schedule. When QA_EMPTY, is all 1s(no QOS to schedule) QW_EMPTY is copied to QA_EMPTY, and corresponding bits QW_EMPTY are made 1. Copying is done only for round robin priority QOSs. Strict priority QOSs will always remain in waiting list. At any point, when the port is to be serviced, based on the priority allocation for that port, first checked to find out if there are any strict priority QOSs are not empty in waiting list. If so then these QOSs are served first. Only when all strict priority QOSs are empty, then round robin QOSs are selected from active list. At any time if eop is sampled for the current QOS, this QOS will be served till eop irrespective of priority. Following algorithm explains how to select QOS for servicing once port is selected.

```
if (prev_qos_v == valid) then current_qos = prev_qos
else if (priority[0] & ~qw_empty[0]) current_qos = 0
else if (priority[1] & ~qw_empty[1]) current_qos = 1
else if (priority[2] & ~qw_empty[2]) current_qos = 2
else if (priority[3] & ~qw_empty[3]) current_qos = 3
else if (priority[4] & ~qw_empty[4]) current_qos = 4
else if (priority[5] & ~qw_empty[5]) current_qos = 5
else if (priority[6] & ~qw_empty[6]) current_qos = 6
else if (priority[7] & ~qw_empty[7]) current_qos = 7
else if (weight_mf  != 0  & qa_empty[active_ptr] = 0)
current_qos = active_ptr
```

```
else if (&qa_empty[7:0]=0)
        // repeat these steps for 0 to 7
    case (prev_qos)  // synopsys parallel_case
     000:
      case(1'b0)
        qa_empty[1] : current_qos = 1
        qa_empty[2] : current_qos = 2
        qa_empty[3] : current_qos = 3
        qa_empty[4] : current_qos = 4
        qa_empty[5] : current_qos = 5
        qa_empty[6] : current_qos = 6
        qa_empty[7] : current_qos = 7
        qa_empty[0] : current_qos = 0
```

To load pending QOSs from waiting to active. This function is performed any time active list is empty. Note that only non_strict priority QOSs are loaded to active list and strict priority QOSs are always serviced from waiting list. To load non priority QOSs to active list use

```
 if (&qa_empty[7:0] = 1)
     qa_empty[0] = priority[0]  | (~priority[0] & qw_empty)
     qw_empty[0] = ~prioirty[0] | (priority[0] & qw_empty[0]);
```

Levels of scheduling

Once a port is selected according to the port calendar, a QOS is selected between all the QOSs that belong to the port. Once a QOS is selected a FlowID is selected between the FlowIDs that belong to that QOS. See Figure 382.

Weight Quota

The weight of a QOS is calculated based on two values. The first value is the weight parameter, which is stored per QOS in the QOS descriptor memory and the second parameter is the weight_quota, which is a programmable vale per all the QOSs. The weight of a QOS is the multiplication of the weight and the weight_quota.

Input Algorithm

1. *Get FlowID and QOS from the shaper/DBS.*
2. *Read QOS memory:*
   - *F_WP = QoS_mem[F_QOS][F_WP]*
   - *F_EMPTY = QoS_mem[F_QOS][F_EMPTY]*
   - *Q_NUM = QoS_mem[F_QOS][Q_NUM]*
   - *PORT_ID = QoS_mem[F_QOS][PORT_ID]*
3. *Read Port memory:*
   - *QA_EMPTY, QW_EMPTY = port_mem[Port_ID][QA_EMPTY, QW_EMPTY]*
   - *PRIORITY = port_mem[Port_ID][PRIORITY]*
4. *Write to memories:*
   // Link FlowID to previous one
   - *Write flow_mem[F_WP][F_NEXT] <= FlowID*

   // Update tail of the link list of the FlowIDs
   - *Write QoS_mem[F_QOS][F_EMPTY] <= 0*

- *Write QoS_mem[F_QOS][F_WP] <= FlowID*
- *If (F_EMPTY == 1) Write QoS_mem[F_QOS][F_RP] <= FlowID*

// Mark the QOS not empty in the port if needed
If (QA_EMPTY[QOS_NUM]==1)&(QW_EMPTY[QOS_NUM] == 1)
- *Write port_mem[PORT_ID][QW_EMPTY[QOS_NUM]] <= 0*

// Link port to calendar in case of Ingress chip configuration
If ((MODE==1) & (|QA_EMPTY) & (|QW_EMPTY)) begin
Write port_mem[INGRESS_PTR[15:8]] <= PORT_ID
INGRESS_PTR[15:8] <= PORT_ID
If(INGRESS_PTR[16]==0) INGRESS_PTR[7:0] <= PORT_ID
INGRES_PTR[16] = 0
end

Output Algorithm

The algorithm describes here doesn't show the conditions for selecting one of the options in step 9.

1. *Get PortID from Port Calendar.*
2. *Read from port memory.*
   - *All parameters*
3. *If (| (QOS_EMPTY)) == 0) terminate output stage*
   Else continue to step 4
4. *If active list is empty -> Move waiting list to Active list and clear waiting list.*
5. *Select QOS according to priority, WRR and previous QOS in middle of packet indication.*
6. *Read from QOS memory, address is {PortID, QOS}.*
   - *All parameters*
7. *Send to PFQ the F_RP value (This is the scheduled FlowID)*
8. *Read FlowID memory*
   - *All Parameters*
9. *Get port status from Status block.*
10. *Select one of the following options depending on parameters from all memories:*
    - *delink_flowid*
    - *move_flowid_to_end_of_active*
    - *don't_move_flowid_and_update_cell_cnt*
    - *do_nothing_to_flowid*
    - *delink_QoS*
    - *move_QoS_to_end_of_active*
    - *don't_move_QoS_and_update_weight_m*
    - *move_QoS_to_waiting*
    - *do_nothing_to_QoS*
    - *For Ingress chip only:*
      - *move_port_to_end_of_link_list*
      - *delink_port*

11. *Update internal and external memories according to above decision.*

Algorithm to calculate q_weight_m during output phase

```
if (qos_empty)
    q_weight_m <= 25'h0
else if (q_weight_m≤0 & eop)          // move to waiting list
    q_weight_m <= q_weight_m – 1 + (q_weight * weight_quota)
else if(q_weight_m!=25'h1000000)      // not the maximum negative value
    q_weight_m <= q_weight_m – 1
```

Memory links for Input Phase

The input phase starts with a FlowID from the DBS. With this parameter all memories are accessed in sequence to get all relevant parameters required for the input phase. See Figure 383.

Memory links for Output Phase

The output phase starts with a port assigned by the port calendar inside the DBS block. The port parameters are read from the internal memory, and a QOS is selected for that port based on the none-empty QOSs and the strict priority. The address to the QOS memory is {PortID, QOS_SELECTED}. The read pointer in the QOS parameters is the FlowID to send to the PFQ. Once reading the parameters of that FlowID it can be sent to the PFQ together with the EOP indication. See Figures 384 and 385.

CPU Port

CPU port is different then the rest of the 64 ports supported. The CPU port does not have any QOS and it's rate is fixed to OC3. FlowIDs are linked to the CPU ports using the external NEXT memory as the next field. FlowIDs are scheduled based on a simple round robin between them.

Blocks Description

The scheduler is divided to several different functional blocks as follow:

Input Phase Block

The input phase contains a small FIFO to accumulate the link requests from the shaper. These link requests are served once an input slot is available. During this stage a FlowID is linked in a link list and if needed a QOS is linked to the waiting lists. Also the port associated with that FlowID is marked as non-empty.

Output Phase Block

The output phase block receives a port from the port calendar. Based on the status of that port, a decision is made if to go ahead with scheduling (if the port is not full) or to dedicate the bandwidth to the software (if the port is full). If scheduling is ok, a QOS is selected according to a weighted-round-robin algorithm and based on that QOS a FlowID is selected. This FlowID is then transferred to the PFQ.

If the cell count of a FlowID becomes zero, means a full packet was sent, the FlowID would be de-linked from the link list if there is no other packets linked to this FlowID. It is possible that because of that the QOS will be de-linked as well.

Software Interface

Software needs to access the scheduler database for several purposes:

1. Initialization after reset: port calendar and QOS databases as well as internal registers.
2. When setting-up a connection, to write the FlowID parameters.
3. For testing purposes of the internal memories mainly.

All software accesses to the scheduler are done through specific registers during the dedicated time slots.

Memory Description

The following is a description of the memories needed to implement the scheduling algorithm.

External Memory – Next FlowID

Address: Flow ID, Length: 1M (20 bits wide)

See Figure 386.

Internal memory – QOS Parameters

Address: QOS, Length: 1K (17 bits wide)

See Figure 387.

Internal memory – QOS Descriptors

Address: QOS, Length: 1K (66 bits wide)

See Figure 388.

Internal memory – Port Parameters

Address: Port ID, Length: 64 (21 bits wide)

See Figure 389.

Internal memory – Port descriptor

Address: Port ID, Length: 64 (47 bits)

See Figure 390.

Internal memory – Next Port

Address: Port ID, Length: 64 (8 bits)

See Figure 391.

Software Interface

The scheduler can be programmed via the software interface by a CPU. All the scheduler's control registers and memories can also be written and read for test purposes through the software interface. The following sections list all the control registers and all the scheduler software interface commands.

Registers

See Figure 392.

Commands

Read FID NEXT Memory

See Figure 393.

Write FID NEXT Memory

See Figure 394.

Read QOS Parameters

See Figure 395.

Write QOS Parameters

See Figure 396.

Read QOS Descriptor

See Figure 397.

Write QOS Descriptor

See Figure 398.

Read PORT Parameters

See Figure 399.

Write PORT Parameters

See Figure 400.

Read PORT Descriptor

See Figure 401.
PQ[0] = PREV_QOS_VALID
PQ[3:1] = PREV_QOS[2:0]
PQ[6:4] = ACTIVE_PTR[2:0]

Write PORT Descriptor

See Figure 402.

Read PORT Next

See Figure 403.

Write PORT Next

See Figure 404.

Init QOS

See Figure 405.

This command write the parameters to the QOS Parameter Memory and also initiates a write to the QOS Descriptor Memory with the following values:
F_EMPTY = 1
Rest of the fields should be reset to zero

Init port

See Figure 406.

This command write the parameters to the Port Parameter Memory and also initiates a write to the Port Descriptor Memory with the following values:
QW_EMPTY[7:0] and QA_EMPTY[7:0] = 8'b1111_1111
Rest of the fields are reset to zero.

Reserved Op-codesReserved Op-codesReserved Op-codes

The op-codes 1111 are reserved.

Timing Diagrams

Global Timing for Input/Output Stages

The input phase and the output phase should be combined during the 8 cycles budget per cell. Since we do so, there are data dependencies between input phase and output phase. Figure 407 shows the memory accesses needed for each memory for every 8 cycles of input/output phase (Figure 408 shows the addresses, all data is 2 cycles later for the external memory):

Input/Output stage Pipes

Input Stage

The input stage actually starts one cycle before cycle1 by sampling a valid signal from the DBS that holds for the entire input stage cycles. If there is no valid signal asserted, the input pipe is frozen by not changing the first pipe stage (this will also save power). See Figure 408.

Output Stage

When working as an egress chip, the output stage starts with reading the port memory parameters using a port ID sent by the port calendar inside the DBS. When in an

ingress chip mode the location to read from is determined by the INGRESS_PTR, which is a register controlled by the scheduler.

The output stage diagram doesn't show the data dependencies to the input phase. Data dependencies will be discussed in the next section. See Figure 409.

Data Dependencies

Data dependencies between the output stage and the input stage

In the scheduler, the data dependencies might happen when the input stage and the output stage access to the same memory address. If one stage reads a memory location before the other finishes updating the same memory location, then the read result might be stale. Or if both stages write the same memory address, the later write might overwrite the previous write result.

There is no data dependency issue for the FlowID memory since the input and output stages are mutually exclusive on a specific FlowID, i.e. the input and output stages never work on the same FlowID. If the input stage is linking a FlowID into the linked list, the output stage is not aware of the existence of the FlowID yet so there is not output phase for this FlowID. If the output stage is scheduling a FlowID, the input stage can not link the same FlowID by the principle of operation. Even if by looking at the FID Next memory operations, there is no overlapping between the input stage and the output stage memory accesses.

For the QOS memory accesses, the input stage only updates the field of F_WP at cycle 7. But the output stage may also update the F_WP field at cycle 6 of the next 8 cycle slot. To resolve the data dependency issue, when the QOS address is the same for the input and the output stages, the input stage updates not only the memory field of F_WP at cycle 7 but also the register field of F_WP which stores the output stage read result of cycle3.

By the same token in the port memory case, when the input and the output stages access to the same port memory address, the input stage updates the memory field of QW_EMPTY at cycle 7 and at the same time it has to update the same QW_EMPTY field of a register which stores the output stage read result at cycle 2 to avoid the data dependency.

The output stage data dependencies between consecutive 8 cycle time slots

The port memory accesses of the output stage take more that 8 cycles, which implies that the port memory accesses in one time slot overlaps the port memory accesses in the following time slot. More specifically, before the previous 8 cycle time slot updates the port memory at cycle 6, the current 8 cycle slot has to read the port memory at cycle 5.

To avoid the data dependencies when the consecutive 8 cycle time slots access to the same port (there is only one port in this case), the read result has to be taken from a register which stores the write date of the previous slot memory write instead of the memory content.

Traffic Shaper/Meter

This block can operate in 2 modes:

- Traffic Shaper mode: in this mode the Shaper shapes specific rates to FlowIDs according to Single or Dual Leaky Bucket algorithm. The shaper selects a Peak rate or a Sustained rate per FlowID depending on an accumulated credit. Once a FlowID has been shaped, it is marked as such in a common.
- Traffic Meter mode: in this mode the Meter measures the rate of the incoming FlowID traffic and mark the FlowID as a violator if the rate is above a specific threshold which is programmed for a FlowID. The user periodically read the indications in the FlowID memory.

See Figure 410.

4.1 Features

1. This block operates at 200MHz clock
2. A FlowID can be issued from the Database at max once every 8 cycles of 200MHz clock.
3. A FlowID is driven to the Database at max once every 8 cycles of 200MHz clock.
4. Shapes input traffic per FlowID using Single or Dual-Leaky-Bucket algorithm.
5. Shapes traffic for rates from ~48.8kbps up to 9.920Gbps.
6. Meter traffic from 12.8kbps up to 9.920Gbps in increments of 64kbps.
7. Up to 1024 different programmable rates can be shaped simultaneously. Each shaped FlowID will point to one of the entries in this table for a specific shaping rate.
8. Up to 1024 different programmable thresholds can be metered simultaneously. Each metered FlowID will point to one of the entries in this table for a specific threshold rate.
9. Each FlowID is programmable to have Dual Leaky Bucket, Single Leaky Bucket or no shaping at all.
10. Can shape or meter up to 1M FlowIDs simultaneously.
11. Supports microprocessor programming of the memories for FID, rates, etc.
12. 28 bits of the external memory in FID1 are used for dual leaky bucket algorithm only. It is possible not to use these bits if using single leaky bucket only. Note that for a Meter function the extra memory for the dual-leaky-bucket is not needed.

See Figure 411 and 412. Note that the memory bits, which are not used for the single leaky bucket should be tied to zero.

See Figure 413. This block doesn't need all the memories for every single mode of operation. Refer to the memory description chapter to see the usage of the memories in each mode.

Functional Description

The traffic shaper receives an input phase from the Database block along with the RateID and then starts shaping the FlowID on a per cell basis. Each time a cell is shaped the Shaper generates a message to the Database. The Database then responds with an empty indication. All packets link list and output to the PFQ are done in the Database block.

Leaky-Bucket Algorithm

The basic leaky-bucket algorithm is an accumulation of credits for the given FlowID, once the amount of credit is enough for a cell, this cell is considered shaped and is transferred. The equation to calculate is:

$$\mathbf{R_{out} = R_{in} * M/N}$$

When $\mathbf{R_{in}}$ is the input rate (based on the selected class), **M** input credit, **N** output credit. The following Figure shows an example of M/N = 2/5:

See Figure 414

.

Dual-Leaky-Bucket Algorithm

A dual-leaky-bucket contains 2 virtual buckets per FlowID (virtual because credit is accumulated only in one bucket). One bucket shapes the traffic for a sustained data rate and the other shapes the traffic for a peak data rate.

The shaped traffic starts by using the $\mathbf{M_s/N_s}$ values of the sustained data rate. If for some reason the accumulated credits are above a specific threshold amount, then the shaped traffic uses the $\mathbf{M_p/N_p}$ values of the peak data rate until the amount of credits returns to zero.

The credit is calculated using the current time measurement. When shaping a cell, the difference between the last shaping time and the current shaping is time calculated to create $\Delta\mathbf{T}$ this is the elapsed time from the last shaped cell. Δ**Credit** is calculated as follow:

$$\mathbf{\Delta Credit = \Delta T * M_s - N_s}$$

$$\mathbf{New\_Credit = Old\_Credit + \Delta Credit}$$

If $\Delta\mathbf{T}$ is long enough then a positive credit is accumulated. If $\Delta\mathbf{T}$ is short then a negative credit is accumulated. Once **New_Credit** is above the threshold then the shaping is done with the peak rate while the credit is always calculated with the sustain data rate.

Implementing Timing Wheel for leaky-bucket algorithm

The timing wheel contains 256K time slots. The wheel is rotating in a rate of one time slot every cell cycle. Each time slot can have few FlowIDs link to it. There is one exit point in the timing wheel, which is time zero. A FlowID is linked to a future time slot based on its shaping parameters. Once that time slot reached time zero, all the FlowIDs that are linked to that specific time slot are dropped to an output FIFO. When a FlowID enters the output FIFO it means that one cell for that specific FlowID is shaped.

Once every cell cycle one FlowID is taken from the output FIFO and is being serviced, means, a new time slot is calculated for the FlowID based on the rateID and some other parameters. The FlowID then is linked to that future time slot. In parallel the shaper sends this FlowID to the Database block, in this block a cell count and packets link list are maintained per FlowID.

It is possible that several FlowIDs are linked to the same time slot. When this time slot arrives to time zero, several FlowIDs are scheduled to the output FIFO at the same cell cycle. In this case the wheel services only one FlowID when the rest are waiting in an output FIFO to be serviced in the next cell cycle. See Figure 415.

The timing wheel is actually fixed and only the zero reference point is moving. That means the wheel can be implemented as a memory and the zero point is a moving address, which is incremented once every cell cycle.
A single leaky bucket algorithm is implemented using the timing wheel. As described above, the Equation to implement the Leaky Bucket is:

$$R_{out} = R_{in} * M/N$$

When assigning K to **N/M**, we will get

$$R_{out} = R_{in} / K$$

The K parameter is the time in the future to place the FlowID. For example if K=3 means the R_{out} is 1/3 of R_{in} then the FlowID is linked to the time slot which it's distance from the current time is 3. Since the timing wheel is rotating once every cell cycle (once every 8 cycles) the next time the FlowID will be seen is after 3 cell cycles when the time slot will reach zero:
See Figure 416.

Fractional value of K

It is possible that K will have a fractional value and an integer value for example to get 40% of the bandwidth K should have the value of 2.5. It is impossible to have a fractional time slot, therefore, to place a FlowID in a future time slot only the integer part of K is used. The fractional part of K is getting accumulated and is added to the integer part only if it is greater or equal to 1. For example, if K=1.28 (78.125% of the line rate) the shaping will appear as in Figure 417.

Output FIFO in the timing wheel

The output FIFO is a link list of the FlowIDs that are shaped for one cell and are waiting to be re-linked to the wheel to some future timing slot. The output FIFO is used because it is possible that more than one FlowID will be linked to a specific slot. In that case, when the slot arrives to time 0, several FlowIDs are shaped at the same time. Since the Shaper can handle only one FlowID every cell cycle, the rest of the FlowIDs have to wait.

Each time a new list of FlowIDs are linked to the output FIFO, they are marked with the time arrival to the output FIFO. When calculating the future time, the result is decreased by the amount of time the FlowID was in the output FIFO. For example, 2 FlowIDs are linked in the output FIFO, K_{FID1}=3 and K_{FID2}=5. The scheduled time slots will appear as in Figure 418.

Dual-Leaky-Bucket using timing wheel

To create a dual-leaky-bucket in timing wheel there is a need for the following parameters:

- K_s for sustain data rate.
- K_p for the peak data rate.
- M_s, M_p Sustain and Peak input rates.
- **Threshold** the threshold value of the accumulated credits to switch to peak rate.
- **Credit** the total amount of positive credits (always a positive number).
- **Residue** the accumulation of the fractional part of K_s or K_p.
- **Time**$_A$ the time of arrival to the output FIFO.

- **$Time_L$** time of last scheduled cell to a time slot.
- **Back Ground Process** is used to mark timer wrapping around since measuring **ΔT**.

There are 3 processes for a FlowID: input phase, output phase and a background process:

- Input phase is when the FlowID is first linked to the timing wheel, same calculation as the output phase is applied except the residue in this case is 0.
- Output phase is when the FlowID is services in the output FIFO.
- Background process is a round robin service between all FlowIDs (1M of them) to mark the wrapping around of the current time. This is done for a FlowID that becomes empty and after a while becomes non empty, the credits should be calculated properly for the FlowID.

When changing the sustain rate to a peak rate then the peak rate is used until the amount of credits is down to zero. The calculation of ΔCredit is being done using the sustain data rate always. Figure 419 describes the use of sustain and peak rates.

Calculation of *ΔCredit*

The calculation of the **ΔCredit** depends of the rates and the time elapsed from the last output phase. There is a small difference between the calculations in the output or the input phases.

K is a pure number. **K** multiple by one time tick gives the amount of time ticks needed to shape one cell. The equation is as follow:

$$\Delta Credit = (\Delta T - 1\ TimeTick * K) / K = (\Delta T - K) / K$$

ΔT is the time elapsed from the last output phase. **K** is decremented from **ΔT** since during that time a cell was shaped. The division by **K** allows measuring the credits in terms of cells. Note that the **K** in the division is the pure number (which is N/M) when the **K** inside the parenthesis has units of time.

- When the output phase uses the sustain or peak rates the equation to be used is:

$$\Delta Credit = (\Delta T - K_s) / K_s = \Delta T/K_s - 1$$

- When the input phase calculates the change in the credit there is no cell shaped therefore:

$$\Delta Credit = \Delta T/K_s$$

Equations (Algorithm) for Input phase

1. *NewCredit = 0*
2. *TimeSlot = INT(K_s) // integer part of K_s*
3. *NewResidue = FRAC(K_s) // fractional part of K_s*

Equations (Algorithm) for Output phase

1. *ΔT = CurrentTime-$Time_L$*
2. *Calculate ΔCredit*
3. *TempCredit = Credit + ΔCredit*
4. *If(TempCredit>Threshold)*

NewCredit = Threshold
Select K = K_p
Else
NewCredit = TempCredit
Select K = K_s

5. *NewTime = (INT(K + Residue) –$Time_A$*
6. *TimeSlot = if(NewTime<0) ? 0 else NewTime*
7. *NewResidue = FRAC(K+Residue)*
8. *NewMarkBits = 11*

Equations (Algorithm) for Background phase

1. *if(MarkBits==11) NewMarkBits = 10*
2. *else if (MarkBits==10) NewMarkBits = 0*

R_{in} and K factor

During input phase or output phase a future time slot is calculated according to K and time variables. A slow rate, means a high K value, will cause a linking of the FlowID to a far timing slot. Whereas a high rate, means a low K value, will cause a linking of a FlowID to a closer timing slot. This chapter is to describe the exact R_{in} for the calculation of:

$$\mathbf{R_{out} = R_{in} / K}$$

The chip runs at 200MHz and the data path is 64 bits, therefore,

$$\underline{\mathbf{R_{in} = 200\ MHz * 64bits = 12.8\ Gbps}}$$

There are 256K timing slots in the timing wheel, therefore, the slowest rate that it can process is:

$$\mathbf{R_{out}(min) = 12.8G/256K = 48.828125\ kbps}$$

In order to have a shaped rate of 64kbps, K should be 200,000:

$$\mathbf{64kbps = 12.8G/200000}$$

That means that if a FlowID is placed in the 200000th time slot in the future each time, its output rate is exactly 64kbps. Figure 420 shows some of the standard rates with their corresponding K. Note that the ratio between OC192 (9.92G) to 12.8G is 1.290

Violator indication

When using a dual-leaky-bucket algorithm the violator indication should be set when using the peak rate. When using a single-leaky-bucket algorithm, the violator indication should not be set.

Metering

Metering is the other mode of operation of this block. The user can assign one of 1K different rates pre defined by the user to a specific FlowID and the block will monitor the traffic to that FlowID. If the actual traffic rate of the FlowID, the measured rate, is greater then the assigned rate then the FlowID is marked by the hardware.
The user should periodically read the marking bit per active FlowID. Once the user reads the FlowID information, the bit is cleared and remains clear until the input traffic is violating again.

External memory for Metering mode

The meter block used the same memory bits of the FlowID but for different meaning. Each FlowID needs to have a counter to count number of incoming cells. Also each FlowID needs to have a bit to mark the FlowID as a violator of the assigned rate or not.

Background process of Metering

The background process of the metering is used to check for the accumulated amount of cells per FlowID every specific amount of time. Once every 8 cycles the background process services one FlowID. That means, a FlowID is being checked by the background process once every 40mSec (at 200MHz clock).

To provide more accuracy to lower rates, a counter is added to each FlowID to control the amount of times the FlowID is accessed by the background process before the actual measurement and marking is done. When the background process accesses a FlowID and this counter is greater then zero, then the background process only decrement that counter, when the counter is zero the background process compares the accumulated amount of cells to the threshold, mark the FlowID is the threshold is smaller then the accumulated amount and then the cell counter is reset, the background counter is set to the value assign by the user.

Once the metering algorithm marked a FlowID as a violator, this FlowID is to remain marked until the user reads the FlowID memory. When the user reads the FlowID memory, the mark bit is reset by the hardware.

Output FIFO for Metering

During the input phase of a FlowID in the metering mode, the FlowID can be marked as a violator. If it is to be marked as a violator, then the FlowID is linked to a link list of violators. The software accesses this list when it wants to get the list of the violators, this way the software doesn't have to read all FlowIDs to detect the violators. Once the FlowID is linked to that list, only when the software reads it, it will be removed from the list. The software first read the value of the MARK_RP register, which is the pointer to the head of the violators link list and then read the FlowID memory. The Meter generates a write to the FlowID memory to clear the mark bit,

Block Description

Shaper's Input Phase

The Shaper input phase receives a rateID, FlowID and valid signal from the DataBase block. This is a request from the database block to shape the FlowID. Once the FlowID is inside the shaper, no more input phase is initiated by the DBS block.
If the specific time slot was empty then it is marked as non-empty, if the specific time slot was not empty then the FlowID is linked to the last FlowID in the link list.

Shaper's Slot Background process

The Traffic Shaper Wheel is constantly rotating one time slot every 8 clocks of the 200MHz clock. As each time slot becomes current, the background process is activated to get al the FlowIDs from the current time slot and link them to the output

FIFO link list. While linking the FlowIDs from the time slot to the output FIFO, the time slot is marked as empty and become the last time position in the wheel.

Shaper's FlowID Background process

This background process is to mark the time in each FlowID such that in the next output phase a ΔT is calculated correctly even if the free running time counter was wrapped around. The process resets the bits while the output phase sets the bits. Each FlowID is serviced once every 1M cell cycles (8M clock cycles). In order to mark a FlowID to use only the threshold value as a credit without using ΔT, this process has to be repeated 2 times. Therefore, a FlowID is marked as "too old to calculate credit" at least after 1M+1 cell cycles and no more then 2M cell cycles (16M clock cycles). In this case the credit value becomes the value of the threshold.

Shaper's Output Phase

The output phase operates once every cell cycle on the FlowID, which is the head of the output FIFO's link list. Operating on a specific FlowID means shaping one cell of the FlowID. The shaper drives the FlowID, which is the head of the output FIFO's link list to the database block. The database block manages the link list of the packets pr FlowID. The act of driving a FlowID to the database block means that one cell is shaped for that FlowID.

The database can return a message to the shaper that the current FlowID is empty, in this case the shaper will not link the FlowID to a future time slot. Note that the Shaper always shapes a cell at a time for packets traffic and for cells traffic. It is the Database block that keeps track of shaped packets/cells and generate de-queue commands to the PFQ accordingly. During metering mode, the output phase is disabled.

Shaper's Time measurement

For the dual-leaky-bucket algorithm there is a need to measure time for calculating the change in the credit. A 21 bits register is used to count the time, this counter is incremented once every 8 cycles. For all calculations the logic used only 18 bits of the time counter except for the FlowID background process.

Since there are 1M FlowIDs supported there is a need to measure time up to 1M units and more, that's is why the time counter has 21 bits.

Traffic Shaper Algorithms

The below algorithms are implementation specific and not only the mathematical equations and conditions. To understand the algorithms, one should look at the memory description first. Memory description.

Shaper Input Phase Algorithm (for single leaky bucket)

1. *Receive from database block: VALID, FID, RATEID*
2. ***RD shp_rate_id_mem[RATEID]{Ks, Kp, Threshold, M}***
3. ***RD FID2_mem[FID]{start, bg_stt, time_msb}***
   RD FID1_mem[FID]{last_time, residue, credit, p_s}
4. *// Calculate future time slot and residue based on previous values*
   if((p_s==1) & bg_stt[1:0]!=2'b00)

```
            a) slot_id[17:0] <= cur_slot_addr[17:0] + INT(Kp)
            b) n_residue[17:0] <= FRAC(Kp)
            c) if(VALID & INT(Kp)==0) link_to_out_fifo <= 1 else link_to_out_fifo <=
               0
        else
            a) slot_id[17:0] <= cur_slot_addr[17:0] + INT(Ks)
            b) n_residue[17:0] <= FRAC(Ks)
            c) if(VALID & INT(Ks)==0) link_to_out_fifo <= 1 else link_to_out_fifo <=
               0
    5. RD slot_mem[slot_id]{rp_slot, wp_slot, e_slot}
    6. // Calculate new credit, see block diagram below
            n_credit = f(current_time, last_time, Ks, M, credit, threshold, bg_stt)
    7. // Calculate rate selector for next output phase
       if(n_credit>=threshold) n_p_s <= 1
       else if(n_credit==0) n_p_s <= 0
       n_p_s <= p_s
    8. // Create link FID to future time slot
    9. if (e_slot==1)
            a) {n_rp_slot, n_wp_slot, n_e_slot} <= FID, FID, 0
            b) n_start <=1
    10. else
            a) {n_rp_slot, n_wp_slot, n_e_slot} <= rp_slot, FID, 0
            b) n_start <= 0
    11. if(link_to_out_fifo)
       Link FID to output FIFO
    12. else if (VALID==1)
       WR slot_mem[slot_id]{rp_slot, wp_slot, e_slot}<= n_rp_slot, n_wp_slot,
       n_e_slot
    13. // Write back for FID fields
    if(VALID==1)
            FID1_word_en[0], Wr FID1_mem[FID]{residue}  <= n_residue
            FID1_word_en[3], Wr FID1_mem[FID]{credit, p_s} <= n_credit, n_p_s
            FID2_word_en[0], Wr FID2_mem[FID]{start}     <= n_start
    if (VALID==1 & e_slot==0) // to avoid a write to an unknown FID's next field
            FID2_word_en[1], WR FID2_mem[wp_slot]{next_FID, bg_stt,
            time_msb}     <= FID, 11, cur_time_reg[20]
Slot background Phase Algorithm
    1. current_slot <= current_slot + 1;          //wraps around 256k
    2. RD shp_slot_mem[current_slot]{rp_slot, wp_slot, e_slot}
    3. WR shp_slot_mem[current_slot]{e_slot} <= 1'b1
    4. If (e_slot==1)
        a. do nothing          //nothing to add to the output registers
    5. else
        a. wp_out_reg <= wp_slot
        b. wr_en_shp_FID_next_mem <= (e_out_reg==0)
        c. if (e_out_reg==0)          //output fifo is not empty
```

i. *rp_out_reg <= rp_out_reg*
ii. *e_out_reg <= e_out_reg*

d. *else*
i. *rp_out_reg <= rp_slot*
ii. *e_out_reg <= 0*

6. *If (wr_en_shp_FID_next_mem==1)*
FID2_word_en[0], WR shp_FID_next_mem[wp_out_reg]{next_FID} <= rp_slot
7. *If(e_slot==0)*
FID1_word_en[1], WR FID1_mem[rp_slot]{arrival_time} <= current_time

FID Background Phase Algorithm (for dual-leaky-bucket only)

1. *current_FID <= current_FID + 1;*
2. ***RD FID2_mem[current_FID]{empty, bg_stat, time_msb, start}***
3. *case (bg_stat) //synopsys full case parallel case*
a. *2'b11: n_bg_stat <= 2'b10*
b. *2'b10: n_bg_stat <= 2'b00*
c. *default: 2'b00*

endcase

4. ***FID2_word_en[1], WR FID2_mem[current_FID]{empty, bg_stat, time_msb, start}<=***
n_empty, n_bg_stat, time_msb, start

Shaper Output Phase Algorithm

1. ***RD FID1_mem[rp_out_reg]{residue, arrival_time, last_time, credit, p_s}***
2. ***RD FID2_mem[rp_out_reg]{next_FID, bg_stt, time_msb, start}***
3. *Send to Database block: rp_out (as O_FID), ~e_out_reg (as O_VALID)*
4. *Receive RateID from Database*
5. ***RD rateid_mem[RateID]{Ks, Kp, threshold}***
6. *// Calculate future time slot and residue based on previous values*
time_spent_in_fifo[17:0] <= current_time[17:0] – arrival_time[17:0]
if(p_s==1)
a) *slot_id[18:0] <= cur_slot_addr[17:0] + INT(Kp) -*
time_spent_in_fifo[17:0]
b) *n_residue[17:0] <= FRAC(Kp)*
c) *if(VALID & INT(Kp)==0) link_to_out_fifo <= 1 else link_to_out_fifo <=*
0

else
d) *slot_id[18:0] <= cur_slot_addr[17:0] + INT(Ks) -*
time_spent_in_fifo[17:0]
e) *n_residue[17:0] <= FRAC(Ks)*
a. *if(VALID & INT(Ks)==0) link_to_out_fifo <= 1 else link_to_out_fifo <=*
0

7. *// Calculate new credit*
a. *n_credit = f(current_time, last_time, residue, Ks, Kp, M, credit, threshold)*
8. *// Calculate rate selector for next output phase*
if(n_credit>=threshold) n_p_s <= 1
else if(n_credit==0) n_p_s <= 0
n_p_s <= p_s

```
9. // prepare values for FID link to a slot
   RD slot_mem[slot_id]{rp_slot, wp_slot, e_slot}
   if (e_slot==1)
        a. {n_rp_slot, n_wp_slot, n_e_slot} <= FID, FID, 0
        b. n_start <=1
        c. next_wr_en <= 0
   else
        a. {n_rp_slot, n_wp_slot, n_e_slot} <= rp_slot, FID, 0
        b. n_start <= 0
        c. next_wr_en <= 1
10. Receive EMPTY indication from Database
11. // Link FID to future time slot
   if (O_VALID==1 & ~EMPTY & link_to_output_fifo)
        link FID to output FIFO
   else if(O_VALID==1 & ~EMPTY )
       WR slot_mem[slot_id]{rp_slot, wp_slot, e_slot}<= n_rp_slot, n_wp_slot,
   n_e_slot
12. // Write back all FID variables
   if (O_VALID & ~EMPTY)
   WR FID1_mem[rp_out_reg]{residue} <= n_residue
   WR FID1_mem[rp_out_reg]{last_time} <= current_time_reg
   WR FID1_mem[rp_out_reg]{credit, p_s} <= n_credit, n_p_s
   WR FID2_mem[rp_out_reg]{bg_stt, time_msb, start} <= 11, cur_time_reg[20],
   n_start
   If(next_wr_en==1) WR FID2_mem[wp_slot]{FID_next} <= rp_out_reg
13. if(rp_out_reg==wp_out_reg & don't place FID in output fifo again) e_out_reg
   <= 1
   else e_out_reg <= 0
14. rp_out_reg <= next_FID
```

Translate {Exp, Mantissa} presentation of K to Integer and Fraction

The K factors are represented in the database with mantissa and exponent.

- For Ks and Kp the presentation is {1.mantissa} x 2^{exp}. exp means the number of places to shift the decimal point to the right.
- For inverse of Ks the presentation is {0.mantissa} x 2^{-exp}. exp means the number of places to shift the decimal point to the left.
- Exp can be common for the Ks value and the InvKS value in the internal memory.

The following algorithm is to translate Ks or Kp value to integer and fraction values from a presentation of {exp, mantissa}. This code is used in the input phase and in the output phase. rate_int = INT(K) and rate_frac = FRAC(K):

```
1. case(exp)      // synopsys full case parallel case
        0: rate_int<={17'b0,1}              rate_frac<=man(17:0)
        1: rate_int<={16'b0,1,man(17)}      rate_frac<={man(16:0),1'b0}
        2: rate_int<={15'b0,1,man(17:16)}   rate_frac<={man(15:0),2'b0}
        3: rate_int<={14'b0,1,man(17:15)}   rate_frac<={man(14:0),3'b0}
        4: rate_int<={13'b0,1,man(17:14)}   rate_frac<={man(13:0),4'b0}
```

5: rate_int<={12'b0,1,man(17:13)} rate_frac<={man(12:0),5'b0}
6: rate_int<={11'b0,1,man(17:12)} rate_frac<={man(11:0),6'b0}
7: rate_int<={10'b0,1,man(17:11)} rate_frac<={man(10:0),7'b0}
8: rate_int<={9'b0,1,man(17:10)} rate_frac<={man(9:0),8'b0}
9: rate_int<={8'b0,1,man(17:9)} rate_frac<={man(8:0),9'b0}
10: rate_int<={7'b0,1,man(17:8)} rate_frac<={man(7:0),10'b0}
11: rate_int<={6'b0,1,man(17:7)} rate_frac<={man(6:0),11'b0}
12: rate_int<={5'b0,1,man(17:6)} rate_frac<={man(5:0),12'b0}
13: rate_int<={4'b0,1,man(17:5)} rate_frac<={man(4:0),13'b0}
14: rate_int<={3'b0,1,man(17:4)} rate_frac<={man(3:0),14'b0}
15: rate_int<={2'b0,1,man(17:3)} rate_frac<={man(2:0),15'b0}
16: rate_int<={1'b0,1,man(17:2)} rate_frac<={man(1:0),16'b0}
default: rate_int<={1,man(17:1)} rate_frac<={man(0),17'b0}
// make case 17 and above default to keep, for exp>=17 use 17 only
endcase

2. *INT(K) <= rate_int – 1 FRAC(K) = rate_frac*

Calculation of K based on desired Rout

Since the shaper performs the following equation:

$$R_{out} = R_{in} / K$$

K value can easily be calculated as:

$$K = R_{in}/R_{out}$$

For rates units of [bit/Sec] **K** is calculated like this:

$$K = 12.8*10^9/R_{out}$$

Credit calculation diagram

For input phase the credit calculation depends on the following parameters:
n_credit = f(current_time, last_time, Ks, M, credit, threshold).
For output phase the credit calculation depends on the following parameters:
n_credit = f(current_time, last_time, residue, Ks, Kp, M, credit, threshold).
Both calculations are almost the same, the implementation should be the same for both and only some inputs may change if using it for input phase or output phase.
Output phase calculation is shown in Figure 421.

Meter Algorithms

Input Algorithm

1. *Receive FID, RATEID, VALID, NCELL from database block*
2. ***RD rateid_mem[RATEID]*** *{threshold}*
3. ***RD FID1_mem[FID]*** *{cell_cnt, mark, bg_cnt, bg_cnt_value}*
4. *n_cell_cnt_tmp[27:0] = cell_cnt[26:0] + NCELL[15:0]*
5. *if(n_cell_cnt_tmp[27]==1) n_cell_cnt <= 27'h7ffffff*
   else n_cell_cnt <= n_cell_cnt_tmp[26:0]
6. *n_mark <= mark | (n_cell_cnt_tmp[27:0] >{1'b0, threshold[26:0]})*
7. *n_bg_cnt <= bg_cnt*
8. *if (mark=0 & n_mark==1) // this FID just violated the metering now – link it to viol_reg*
   if (MARK_EMPTY==1)

```
                    {n_MARK_RP, n_MARK_WP, n_MARK_EMPTY} <= FID, FID, 0
                    wr_next_FID <= 0
                else
                    {n_MARK_RP, n_MARK_WP, n_MARK_EMPTY} <= MARK_RP, FID, 0
                    wr_next_FID <=1
        else    // either the FID did not violate metering or it is already marked
                {n_MARK_RP, n_MARK_WP, n_MARK_EMPTY} <= MARK_RP,
                MARK_WP, MARK_EMPTY
                wr_next_FID <= 0
    9. if(VALID==1)
                WR FID1_mem[FID] <= {n_cell_cnt, n_mark, n_bg_cnt, bg_cnt_value}
    10. if(VALID==1 & wr_next_FID==1)
                WR FID2_mem[MARK_WP]{next_FID} <= FID
```

Background Algorithm

```
    1. Read FID_mem[FID] {cell_cnt, mark, bg_cnt, bg_cnt_value}
    2. if(mark==1)  // freeze since the FID is marked
        a. n_bg_cnt <= bg_cnt
        b. n_cell_cnt <= cell_cnt
    3. else if(bg_cnt==0)    // start a new counting cycle
        a. n_bg_cnt <= bg_cnt_value
        b. n_cell_cnt <= 0
    4. else
        a. n_bg_cnt <= bg_cnt - 1
        b. n_cell_cnt <= cell_cnt
    5. Write FID_mem[FID] <= {cell_cnt, mark,  n_bg_cnt, bg_cnt_value}
```

Software read process

```
    1. RD FID1_mem[MARK_RP] {cell_cnt, mark, bg_cnt, bg_cnt_value}
    2. RD FID2_mem[MARK_RP] {next_FID}
    3. n_cell_cnt = 0
    4. n_mark = 0
    5. n_bg_cnt = bg_cnt_value
    6. if (MARK_EMPTY==0)       // some marked FIDs are avail
            if (MARK_RP==MARK_WP)// last FID of marked FID's link list
                n_MARK_RP <= MARK_RP
                n_MARK_EMPTY <= 1
            else                                    // more than one marked FID's avail
                n_MARK_RP <= next_FID
                n_MARK_EMPTY <= 0
        else                    // no marked FID's avail
            n_MARK_RP <= MARK_RP
            n_MARK_EMPTY <= 1
    7. if(VALID==1 & MARK_EMPTY==0)
            WR FID1_mem[MARK_RP] <= {n_cell_cnt, n_mark, n_bg_cnt,
            bg_cnt_value}
```

Calculation of threshold for Meter mode

The meter block can measure the number of cells for different intervals of times depending n the value of the BG_CNT[3:0] as shown in Figure 422,
For each measured output rate there is a maximum amount of cells that can be accumulated in 40mSec amount of time (threshold for 40mSec). The equation is:

Threshold [#Cells/40mSec] = Rout[bits/Sec]/(25*512) = Rout/12800

For example, a rate of 64,000bps means a threshold of 5 cells/40mSec. Since this number is too small, it doesn't enable much of granularity in the measurement, an increase on BG_CNT can help since the measurement will be over longer period of time. If BG_CNT is 15, then the threshold should increase to 5*16 = 80 cells.
The user has the tradeoff between the time to measure and the granularity of the measurement.

Memory description

Figure 423 shows what memories and bits are needed per mode of operation.

Rate ID/Threshold Internal Memory

Shaper Rate ID for Shaper mode

Length: 1024, Address: 10 bit rateID from database block, #bits: 86
See Figure 424.

Meter Threshold for Meter mode

Length: 1024, Address: 10 bit ThrsID from database block, #bits: 27
See Figure 425.

Shaper Slot Memory

Length: 256k, Address: 18 bit slot location, #bits: 41
See Figure 426.

Shaper/Meter FlowID Memory

The FID memory is divided to 2 memory blocks with different address to each. Word write-enable signals are assigned to these memories for the shaper mode.
Meter algorithm uses 36 bits in FID1 memory and no bits in FID2 memory.
Shaper algorithm uses 65 bits in FID1 memory and 25 bits in FID2 memory.
When using single-leaky-bucket only, the unused memory bits should be tied down to zero.

FID MEM1 for Shaper

Length: 1M, Address: 20 bit FlowID, #bits 72

See Figure 427. (If more bits are needed because of future needs, the credit field can be reduced)

FID MEM1 for Meter

Length: 1M, Address: 20 bit FlowID, #bits: 36
See Figure 428. There is no need to word write-enable signals in case of Meter mode.

FID MEM2 for Shaper

Length: 1M, Address: 20 bit FlowID, #bits: 24
See Figure 429.

FID MEM2 for Meter

Length: 1M, Address: 20 bit FlowID, #bits: 20
See Figure 430.

CPU Interface

Registers Description

See Figure 431.

CPU Commands for Shaper mode

Read FID1 memory

See Figure 432.
P – peak_sustain
Registers R2, R3 are used in case of dual-leaky-bucket only.

Write FID1 memory when using shaper

See Figure 433.
P – peak_sustain
Registers R2, R3 are used in case of dual-leaky-bucket only.

Read FID2 Memory

See Figure 434.
SBT[0] – start (bit 20 in the register)
SBT[2:1] – bg_stt[1:0] (bits [22:21] in the register)
SBT[3] – time_msb (bit [23] in the register)

Write FID2 Memory

See Figure 435.
SBT[0] – start (bit 20 in the register)
SBT[2:1] – bg_stt[1:0] (bits [22:21] in the register)
SBT[3] – time_msb (bit [23] in the register)

Read SHP_SLOT memory

See Figure 436.
E – empty indication (bit [20] in the register)

Write SHP_SLOT memory

See Figure 437.
E – empty indication (bit [20] in the register)

Read RATE_ID memory

See Figure 438.
Ks[17:0] – Ks Mantissa, Ks[22:18] – Ks Exponent (same for Kp)

Write RATE_ID memory

See Figure 439.
Ks[17:0] – Ks Mantissa, Ks[22:18] – Ks Exponent (same for Kp)

Setup connection command

See Figure 440. This command resets all fields in FID1 and FID2 memories. Except the back_count field in FID2 memory, which is driven from the command itself. Also bg_stt[1:0] and empty bits should be set to 1 during this command.

CPU Commands for Meter mode

Read FID1 memory

See Figure 441.
M – Mark bit (bit [27] in the register)
Cur[3:0] – bg_cnt
Cnt[3:0] – bg_cnt_value

Write FID1 memory

See Figure 442.

Read FID2 Memory

See Figure 443.

Write FID2 Memory

See Figure 444.

Read Threshold (internal) Memory

See Figure 445.

Write Threshold (internal) Memory

See Figure 446.

ReadModWrite Meter

See Figure 447.

M – Mark bit (bit [27] in the register)
Cur[3:0] – bg_cnt
Cnt[3:0] – bg_cnt_value
This command triggers a write back to the same location clearing all values to zero (except the bg_cnt_value). That's is why this command has a different opcode then a simple read.

Setup connection command

See Figure 448.

This command writes the following data to FID1 memory:
cell_cnt <= 0, mark <= 0, bg_cnt <= cnt[3:0], bg_cnt_value <= cnt[3:0]

Reserved Op-codes

The op-codes 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111 are reserved.

Timing Diagrams for Shaper

The Shaper is able to process one input phase, one background phase and one output phase every 8 clock cycle slots.

Timing Diagrams for Meter

The Meter needs to perform one input phase, one background process and one CPU Rd/WR accesses during 8 cycles budget. Figure 449 is a timing diagram that shows the accesses to the memories during these 8 cycles.

The CPU Rd/Wr operation and the Meter software phase cannot exist together at the same 8 cycles. If a ReadModWrite command is issued then no other CPU interface is possible during these 8 cycles.

For regular Read/Write from the internal/External memories, all accesses occur at slot 7. There are no data dependencies between any of the accesses. The ReadModWrite command clears all fields therefore the background process can be overwritten by it.

MX 2 CPU Interface

This block is responsible for three different functions:

- Connects between the blocks and the CPU via the CPU interface. It decodes the address to Figure which block to access and generate the chip select to each block.
- Generates global synchronization signals to the rest of the blocks.
- Holds general control registers like version register, mode register etc.

Features

- Address bus of 10 bits per chip.
  Bits 0-5 are used as an address inside the block (up to 64 registers per block)
  Bits 9-6 are used to select a block (up to 16 blocks per chip)
- Data bus of 32 bits per chip.
  Externally this bus is b-directional
  Internally this bus is split to 2 busses: DATA_IN and DATA_OUT.
- Chip select signal is the qualifier for data, address and read/write busses.
- The CPU interface can support multiplexed CPU interface or non-multiplexed CPU interface. When using multiplexed interface, the address and data are folded to the DATA[31:0] bus.
- Support little endian and big endian.
- CPU accesses all registers in all blocks via direct read/write commands.
- CPU accesses all internal/external memories in all blocks via indirect accesses using the general-purpose registers dedicated for these accesses.
- Blocks addresses are shown in Figure 450.
- Generates a global sync signal once every 520 cycles of the 200MHz clock (65 ports, 8 cycles per port).
- Multiplexes all test busses from the blocks to one output test bus.

Functional Description

The CPU IF block is used to interface with the CPU. The CPU IF block decodes the incoming address bus to generate a unique chip-select signal to each block. The CPU IF block also holds general registers that contain general parameters and modes for the entire chip. Basically any logic that is not related directly to a specific block is in this block.

The CPU IF block can operate in two modes depending on an input pins to select between them:

- Regular CPU access: separate busses for data and for address. In this case address bus is qualified be the chip-select signal.
- Multiplexed CPU access: one bus is multiplexed for address and for data. In this case address is qualified by the ALE signal.

The CPU interface is based on Intel's microprocessor interface (when multiplexing address/data bus). The following timing diagrams defines the characteristics of the CPU interface for Read and for Write accesses:

Read access characteristics

See Figure 451.

- RDWR_L signal has the same timings as the ADDR bus.
- A valid read cycle is defined when RDWR_L is latched as high on the negative edge of the CS_L signal.
- In non-multiplexed address/data bus architecture, ALE should be held high so parameters t_{Salr}, t_{Halr}, t_{VL}, t_{Slr} and t_{Hlr} are not applicable.
- In our implementation the address is latched on the negative edge of the CS_L therefore t_{Har} is not applicable.

See Figure 452.

Write access characteristics

See Figure 453.

- RDWR_L signal has the same timings as the ADDR bus.
- A valid write cycle is defined when RDWR_L is latched as low on the negative edge of the CS_L signal.
- In non-multiplexed address/data bus architecture, ALE should be held high so parameters t_{Salw}, t_{Halw}, t_{VL}, t_{Slw} and t_{Hlw} are not applicable.
- In our implementation the address is latched on the negative edge of the CS_L therefore t_{Haw} is not applicable.

See Figure 454.

Timing diagram of signals to blocks

The following diagram describes the timings of the CPU interface to each block. Note that chip-select signal is now asserted for only one cycle of the 200MHz clock. For a write or read cycle, all parameters are valid and stable before chip-select is asserted. When a read cycle occurs data is driven from the block after a maximum of 3 clock cycles of 200MHz clock.

See Figure 455.

Configuration modes

The CPU interface has 2 configuration bits to control the CPU interface as shown in Figure 456.

GSYNC - Synchronization signal

GSYNC signal is a one-cycle pulse in the 200MHz clock, which is asserted once every 520 cycles. The SYNC signal is used for synchronization between the blocks and between the chips as well as to generate the slot-count and the CPU-port indication. See Figure 457.

Slot count and CPU port indication

The signals slot_count and CPU_port are generated inside each block base on the GSYNC signal. Each block first needs to sample GSYNC and use the sampled version to generate the signals. See Figure 458.

Test Multiplexing

The CPU interface block receives test busses from each block and multiplex them to one output test bus. The width of all test busses is 32 bits. Each block has a control register to select the information to drive to the pins. The CPU interface block has a control register to select between the blocks.
See Figure 459.

Interface

The CPU IF block generates chip-select signal per block, the rdwr_l, addr and data busses are propagated to the blocks directly. The blocks perform the read or write command based on the rdwr_l signal only when the chip-select signal per the block is asserted. The chip-select signal per each block is synchronous to the 200MHz clock domain already. Each block first samples the incoming signals and then uses them. This is being done to solve any timing issues that may occur. If during placement and routing the signals from the CPU IF to the blocks will violate the 5nSec budget, another set of FFs can be placed in the middle of the path.
For a read cycle the CPU IF block selects the data busses coming in from the blocks based on the chip-select per block (the address).

See Figure 460; software reset and testing signals are not included in the diagram. The address, data and rd/wr signals takes about 4 cycles of 200MHz to propagate from the chip's pins to the block (6 cycles are used for delay calculation). 2 cycles are needed to perform read or write accesses in the block.
Therefore a total of 8 cycles are needed to perform a read or a write access.

Block Descriptions

CPU Interface

The negative edge of the CS_L signal qualifies the ADDR[9:0] and the RDWR_L signals. The positive edge of the CS_L signal qualifies the DATA bus for read and for write accesses. The CS_L is basically an enable signal to latches. Figure 461 shows the basic latching circuit and the synchronization to 200MHz clock domain:

Registers

Figure 462 describes the registers implemented in the CPU IF block. These registers act as control to the rest of the chip. The registers are implemented in the 200MHz clock domain even if there is no need to do so since their value is fixed and stable during normal operation.

Appendix A: Tags for Lookup Engine

The following types of traffic are supported on the Maximus chip Lookup Engine Block:

1. ATM
2. MPLS
    a. ATM
    b. PPP
    c. Ethernet

d. Frame Relay
3. Ethernet
4. IP Frame Relay

(All bit numbering assumes Big Endian ordering)

See Figure 268.

For the first version of the chip, the Flow Descriptor(FD) in Lookup Engine block will support Data Types 1 and 2(a) thru 2(d). Pure Ethernet, IP and Frame Relay will be supported in the next version of the Maximus chip. Header format references for these traffic types are given below:

HEADER FORMATS

1. ATM – Tag in the Link Layer (carried in 28 bit fields - 12 bit VPI/16 bit VCI)
(Computer Networks, 3rd Edition, Andrew Tannenbaum, pg. 451)

See Figure 269. Each field is 8 bits wide with 12 bit VPI and 16 bit VCI field, for a total of 28 bits lookup tag. This 28 bit lookup tag is located in the 1st 64-bit word, bits 1 thru 28.

2(a) MPLS – ATM, 2(b) MPLS-PPP, 2(c) MPLS-Ethernet and 2(d) MPLS-FR
(Tag is always between the Link and the Network Layer)
(MPLS, Technology & Applications, Davie & Rekhter, pg. 132)

The MPLS Header is inserted between the Link Layer and the Network Layer of each of the above four protocols. The detailed Link Layer description of each of the protocols is given below in other sections of this document.

The MPLS field is the same for the four different Link Layer protocols. The exact location of this 20-bit MPLS label in the Frame varies with the Link Layer used, and is summarized in the table above. With 20 bits, the MPLS label allows having 2^{20} possible tags. See Figure 270.

3. Ethernet – Tag in the Link Layer
(Computer Networks, 3rd Edition, Andrew Tannenbaum, pg. 281)

Ethernet Destination Address(DA) field is used in the Lookup Engine Block. These are the 48 bits after the first 64 bits as shown below. See Figure 271.

4. IP – Tag in Network Layer
(Computer Networks, 3rd Edition, Andrew Tannenbaum, pg. 413)

The IP Packet Destination Address is used for Lookup Engine. Starting from the the 3rd 64 bit word, the Destination Address is 32 bits field and is used in the Lookup Engine. This assumes Big Endian Transmission. See Figure 272.

5. Frame Relay (FR) - Tag in Link Layer (carried in 10 bit DLCI field)
(Frame Relay Technology & Practice, Buckwalter, pg. 39) See Figure 273.

6. Point to Point Protocol (PPP)
(Computer Networks, 3rd Edition, Andrew Tannenbaum, pg. 232) See Figure 274.

Appendix B: Flow Control Mechanism of CSIX-L1

CSIX provides multiple levels of flow control and supports flow control in both directions. At ingress, flow control is fabric-to-TM; at egress, flow control is TM-to-fabric.

Link-level Flow Control

The link level flow control is symmetric across the CSIX interface and provides independent control for data and control queues. There are 2 link level queues. One queue is for control traffic and the other queue is for data traffic. For each queue there is a **ready** bit in every header indicating the congestion status for the receive queue of the respective traffic type. See Figure 275.

Fabric Flow Control

The fabric flow control provides a finer level of flow control than link level flow control by specifying the specific TM and Class that are oversubscribed. Flow control on a TM port (line-end or VC) basis is the responsibility of cooperating TMs. Flow control frames can go in both the ingress and egress directions. See Figure 276 and 277.

Appendix C—MX2 Software Interface: A User Guide to Software Interface

This document describes the software interface to the MX2 chip via the CPU interface. This document should include all atomic CPU commands as well as the high levl commands.

There are two types of commands:

1. Low-level commands, when issued, the driver generates a single command to access registers or internal/external memories in one specific block.
   For example read scheduler's external memory
2. High-level commands, when issued, the driver generates several commands to access registers or internal/external memories in several blocks.
   For example setup connection command to initialize FID database in several blocks.

Features

1. the interface supports up to 16 different blocks in the chip, each block has up to 64 registers.
2. CPU accesses all registers in all blocks via direct read/write commands.
3. CPU accesses all internal/external memories in all blocks via indirect accesses using the general-purpose registers dedicated for these accesses.
4. CPU interface can support multiplexed address/data bus or non-multiplexed.
5. CPU interface can support big endian or little endian modes.

Interface

Physical Interface

The CPU interface to the MX2 chip is via standard CPU interface that contains signals as shown in Figure 463. The software, using this interface, can directly access the registers inside each block and can indirectly access internal and external memories per each block.

Address spaces per block/chip

The 4 MSBs of the address bus points to the specific block, while the 6 LSBs of the address points to internal registers inside the block. Each block has address space of 64 entries only. The internal registers of each block are directly mapped to these addresses and external/internal memories are indirectly mapped. The MX2 chip can support up to 16 blocks with 64 registers each. Figure 464 is the allocation of address spaces per block.

Direct Access

A direct access is used to access registers only. The only two commands used in a direct access are read and write. Each register has it's own unique address inside the chip.

Indirect Access

An indirect access is used to access internal or external memories. Each block has a set of registers dedicated for the indirect access. The COM register contains an Op-code and address of the memory to access. Registers R0 through Rn are used to contain the data.

Indirect Write Command

A write command starts with writing the data to registers R0 to Rn and then writing the opcode and the address to the COM register. The opcode defines which memory to access and if it is a read or write command.
The hardware then, takes the data from registers R0 through Rn and write them to a specific address defined in the COM register in the specific memory defined by the opcode.

Once the command is complete, the hardware resets the opcode bits in the COM registers. It is guarantee that aread/write command to internal/external memory will be complete after 12 cycles of the operating clock (@200MHz it means 60nSec).

Indirect Read Command

A read command starts with writing the opcode and the address to the COM register. The opcode defines which memory to access and if it is a read or write command.

The hardware then, uses the address to perform a read access the memory. Once the data arrives the hardware assign the data to registers R0 through Rn and then clears the opcode bits in the COM register.

The software then, generates direct reads to registers R0 through Rn to get the data. It is guarantee that aread/write command to internal/external memory will be complete after 12 cycles of the operating clock (@200MHz it means 60nSec).

CPUIF Block

CPUIF block doesn't have any internal or external memories, therefore, only direct commands to registers are possible in this block.

Registers

See Figure 465.

Commands

N/A

PFQ Block

Registers

See Figure 466.

Commands

Read FID Enqueue Memory (72 bits) – Up to 2M locations

See Figure 467.

Write FID Enqueue Memory (72 bits) – Up to 2M locations

See Figure 468.

Read FID Dequeue Memory (36 bits)– Up to 1M locations

See Figure 469.

Write FID Dequeue Memory (36 bits) – Up to 1M locations

See Figure 470.

Read BID Memory (36 bits) – Up to 16M locations

See Figure 471.

Write BID Memory (36 bits) – Up to 16M locations

See Figure 472.

Read Stat Memory (72 bits) – Up to 2M locations

See Figure 473.

Write Stat Memory (72 bits) – Up to 2M locations

See Figure 474.

Setup connection command for FID

See Figure 475. This command sets up a connection by accessing external memories and initializes the corresponding locations to the proper parameters. It is a write command. Clear PFQ entries for ENQ, DEQ and STAT!

Tear-Down connection command for FID

See Figure 476. This command tears down a connection by accessing external memories. It is a write command. The data is all "0" PFQ returns all remaining BIDS on the queue of the down connection to the free link list during this command.

INIT FID MEMORIES

The memories are the enqueue, dequeue, and statistics memories. See Figure 477.

INIT BID Memory

This will build the free buffer list. See Figure 478.

Reserved Op-codes

The op-codes 1101, 1110, and 1111 are reserved. The following commands are combinations of the above simple commands. A user command like this should be broken to several Read or Write commands in the driver.

Initialized External memories

- Write "0" in all memory locations.
- Write "NULL" in the enqueue memory.
- Write "NULL" in the dequeue memory.
- Build the free buffer list by linking each buffer to the next one.

Initialized Internal memories

- Write "NULL" in the Multicast Root memory.
- Write "NULL" in the Multicast Leaf memory.
- Write "NULL" in the Tunneling Root memory.
- Write "NULL" in the Tunneling Leaf memory.

DataBase Block

Registers

See Figure 479.

Commands

Read external FID memory

See Figure 480.

A[0] – C_P (bit [6] in the register)
A[1] – CPU_PORT (bit [7] in the register)
Pri[2:0] – Priority for shaped traffic
S – Shape (bit [23] in the register)
C[0] – clpi (bit [10] in the register)
C[1] – Empty (bit [11] in the register)

Write external FID memory

See Figure 481.

A[0] – C_P (bit [6] in the register)
A[1] – CPU_PORT (bit [6] in the register)
Pri[2:0] – Priority for shaped traffic
S – Shape (bit [23] in the register)
C[0] – clpi (bit [10] in the register)
C[1] – Empty (bit [11] in the register)

Setup FID connection

See Figure 482.

A[0] – C_P (bit [6] in the register)
A[1] – CPU_PORT (bit [6] in the register)
Pri[2:0] – Priority for shaped traffic
S – Shape (bit [23] in the register)
During the setup connection command, the user defines the above fields only.
The fields RP, WP, NCELL, CELL_CNT and CLP should be reset to 0 by the hardware
The field EMPTY should be set to 1 by the hardware.

Read external PKT memory

See Figure 483.
L[0] – CLP
L[1] - Reserved

Write external PKT memory

See Figure 484.
L[0] – CLP
L[1] - Reserved

Read port calendar memory

See Figure 485.
Data[5:0] – portid
Data[6] – valid
Data[7] – jump

Write port calendar memory

See Figure 486.
Data[5:0] – portid
Data[6] – valid
Data[7] – jump

Read SHP Strict memory

See Figure 487.
P[2:0] – previous QOS pointer
P[3] – previous QOS pointer valid

Write SHP Strict memory

See Figure 488.

Read SHP out memory

See Figure 489.

Write SHP out memory

See Figure 490.

Init SHP_out_port memory

See Figure 491. This command generates write accesses to all entries in the Shp_out memory (64 entries) with a specific data of: p_rp = 0, p_wp = 0, p_e = 1

Init Shp_strict memory

See Figure 492. This command generates write accesses to all entries in the Shp_strict memory (64 entries) with a specific data of: empty = 8'b1111_1111, prev_qos_v = 0, prev_qos = 0

Reserved Opcodes

The opcodes: 1110, 1111 are reserved.

Shaper/Meter Block

Registers

See Figure 493.

Commands for Shaper mode

Read FID1 memory

See Figure 494.
P – peak_sustain
Registers R2, R3 are used in case of dual-leaky-bucket only.

Write FID1 memory when using shaper

See Figure 495.
P – peak_sustain
Registers R2, R3 are used in case of dual-leaky-bucket only.

Read FID2 Memory

See Figure 496.
SBT[0] – start (bit 20 in the register)
SBT[2:1] – bg_stt[1:0] (bits [22:21] in the register)
SBT[3] – time_msb (bit [23] in the register)

Write FID2 Memory

See Figure 497.
SBT[0] – start (bit 20 in the register)
SBT[2:1] – bg_stt[1:0] (bits [22:21] in the register)

SBT[3] – time_msb (bit [23] in the register)

Read SHP_SLOT memory

See Figure 498.
E – empty indication (bit [20] in the register)

Write SHP_SLOT memory

See Figure 499.
E – empty indication (bit [20] in the register)

Read RATE_ID memory

See Figure 500.
Ks[17:0] – Ks Mantissa, Ks[22:18] – Ks Exponent (same for Kp)

Write RATE_ID memory

See Figure 501.
Ks[17:0] – Ks Mantissa, Ks[22:18] – Ks Exponent (same for Kp)

Setup connection command

See Figure 502.
This command resets all fields in FID1 and FID2 memories. Except the back_count field in FID2 memory, which is driven from the command itself.
Also bg_stt[1:0] and empty bits should be set to 1 during this command.

Commands for Meter mode

Read FID1 memory

See Figure 503.
M – Mark bit (bit [27] in the register)
Cur[3:0] – bg_cnt
Cnt[3:0] – bg_cnt_value

Write FID1 memory

See Figure 504.

Read FID2 Memory

See Figure 505.

Write FID2 Memory

See Figure 506.

Read Threshold (internal) Memory

See Figure 507.

Write Threshold (internal) Memory

See Figure 508.

ReadModWrite Meter

See Figure 509.
M – Mark bit (bit [27] in the register)
Cur[3:0] – bg_cnt
Cnt[3:0] – bg_cnt_value
This command triggers a write back to the same location clearing all values to zero (except the bg_cnt_value). That's is why this command has a different opcode then a simple read.

Setup connection command

See Figure 510. This command writes the following data to FID1 memory:
cell_cnt <= 0, mark <= 0, bg_cnt <= cnt[3:0], bg_cnt_value <= cnt[3:0]

Reserved Op-codes

The op-codes 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111 are reserved.

Scheduler Block

Registers
See Figure 511.
Commands

Read FID NEXT Memory

See Figure 512.

Write FID NEXT Memory

See Figure 513.

Read QOS Parameters

See Figure 514.

Write QOS Parameters

See Figure 515.

Read QOS Descriptor

See Figure 516.

Write QOS Descriptor

See Figure 517.

Read PORT Parameters

See Figure 518.

Write PORT Parameters

See Figure 519.

Read PORT Descriptor

See Figure 520.

PQ[0] = PREV_QOS_VALID
PQ[3:1] = PREV_QOS[2:0]
PQ[6:4] = ACTIVE_PTR[2:0]

Write PORT Descriptor

See Figure 521.

Read PORT Next

See Figure 522.

Write PORT Next

See Figure 523.

Init QOS

See Figure 524. This command write the parameters to the QOS Parameter Memory and also initiates a write to the QOS Descriptor Memory with the following values:
F_EMPTY = 1
Rest of the fields should be reset to zero

Init port

See Figure 525. This command write the parameters to the Port Parameter Memory and also initiates a write to the Port Descriptor Memory with the following values:
QW_EMPTY[7:0] and QA_EMPTY[7:0] = 8'b1111_1111
Rest of the fields are reset to zero.

Reserved Op-codesReserved Op-codesReserved Op-codes

The op-codes 1111 are reserved.

Driver's high level commands

The following section describes higher level commands then just read/write to internal/external memories. The following commands are basically combinations of the above commands and are created to give easier user interface to the chip.
These commands doesn't have special opcode in the chip itself, they are implemented in the driver only.

Setup Connection Command

This command sets up a connection by accessing external memories in several blocks and initializes the proper parameters per FlowID.

The following table describes the parameters per FlowID that should be set during the setup connection command, for more specific information about the fields, please refer to the blocks SPECs in Figure 526.

Once the driver receives a setup connection command, the driver generates setup connection command to the DBS, SHP and PFQ blocks. The parameters to the setup connection commands to the blocks are driven from the user along with the global setup connection command to the driver.

Tear-down Connection Command

A tear-down-connection command is used only in the PFQ block. This command is needed to invalidate some fields in the FID descriptor memories as well as the statistics fields. The Database, Shaper and Scheduler blocks don't need to invalidate any of the FID fields. Therefore, there is no special driver command for this command. The user can initiate a simple write command to the PFQ to implement this command.

Power-up initialization

When reset is applied to the chip (or power-up) the internal control registers wakes up in their default value, all traffic is disabled and internal/external memories are not defined. The following is the initialization sequence the user should perform before allowing any traffic to the MX2 chip:

1. PFQ – issue "Init FID Memory" command
2. PFQ – issue "Init BID Memory" command
3. PFQ – initialize multicast internal memories (if needed)
4. PFQ – initialize tunneling internal memory (if needed)
5. PFQ – specify bits in control register if different then default (address 36)
6. PFQ – initialize RED internal memory only if using RED
7. PFQ – initialize class threshold registers (addresses 45 to 52) only when using CLASS
8. DBS – initialize Port Calendar memory
9. DBS – issue "Init Shp_strict Memory" command
10. DBS – issue "Init Shp_out_port Memory" command
11. CPUIF – define Shaper/Meter mode (register 1 bit [8])
12. SHP – initialize RateID/Threshold memory
13. SHP – issue "Init Slot Memory" command
14. SCH – initialize Port Descriptor Memory
15. SCH – initialize QOS descriptor Memory
16. <setup connections if needed>
17. PFQ – enable input traffic for specific ports by writing to registers 43 and 44
18. SCH – enable output traffic from scheduler (register 32 bit [0])
19. SHP – enable output traffic from shaper (register 32 bit [1])

Once these steps are complete, the user can initiate setup connection commands.

The user doesn't have to go through all of the above steps, for example, if tunneling is not used then there is no need to initialize the internal memory for tunneling in the PFQ block.the MX2 Chip

Appendix D—MX2 External Memory

See Figure 527.

While this invention has been described in terms of several preferred embodiments, there are alterations, permutations, and equivalents, which fall within the scope of this invention. It should also be noted that there are many alternative ways of implementing the methods and apparatuses of the present invention. For example, the present invention may be practiced by integrating line card functionality with switch fabric functionality such that a router does not have individual line cards. Alternatively, router in accordance with the invention may include many lines cards deploying the Maximus device, some of the line cards functioning as ingress line cards, others of the line cards functioning as egress line cards. It is therefore intended that the following appended claims be

interpreted as including all such alteration, permutations, and equivalents as fall within the true spirit and scope of the present invention.